Laine Janne ja Ylisiurua Juho

# DATAORIENTOITUNUT SUUNNITTELU JA FUNKTIONAALINEN OHJELMOINTI UNITYSSÄ

Tietojenkäsittelyn
tutkinto

Kevät 2017

KAJAANIN
AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

# TIIVISTELMÄ

**Tekijä(t):** Laine Janne ja Ylisiurua Juho

**Työn nimi:** Dataorientoitunut suunnittelu ja funktionaalinen ohjelmointi Unityssä

**Tutkintonimike:** Tradenomi, tietojenkäsittely

**Asiasanat:** peliohjelmointi, dataorientoitunut suunnittelu, funktionaalinen ohjelmointi, Unity, rinnakkaislaskenta, komponenttimalli

Unity-kehitysympäristö on noussut viime aikoina yhdeksi suosituimmista kehitysvälineistä pelialalla. Pelimoottorin vanhuus ja huonot arkkitehtuuriset ratkaisut ovat kuitenkin aiheuttaneet tilanteen, että Unity ei perustasoltaan ole enää riittävän hyvä pelien kehittämiseen. Tämän opinnäytetyön tavoitteena oli tutkia parempia tapoja ohjelmoida pelejä, kuin nykyaikana on totuttu käyttämään ja soveltaa niiden käyttöä Unityssä ja lisäksi työn on tarkoitus toimia ohjeena ohjelmoijalle, joka haluaa tietoa vaihtoehtoisista ohjelmointitavoista Unityssä.

Tietoperusta koostuu pelikehinkehityksen nykytilasta, painottaen Unityllä kehittämistä, ja kahden ohjelmointimallin, dataorientoituneen suunnittelun ja funktionaalisen ohjelmoinnin, esittelystä. Tietoperustan pohjalta työssä luotiin esimerkki pelin arkkitehtuurista, jossa käytetiin esiteltyjä ohjelmointimalleja.

Teoriaosuus aloitettiin selvittämällä mitä haasteita pelinkehitys nykyään sisältää, mitä ongelmia nykyisissä ohjelmointikäytänteissä on ja mitä niissä pitäisi korjata. Ongelmien ratkaisuun käytettävistä ohjelmointimalleista esitellään teoria ja näytetään käyttökohteita kuvallisin esimerkein. Tämän lisäksi molempien mallien hyvät ja huonot puolet käsitellään. Työn viimeisessä osiossa testataan käytännössä esiteltyjen ohjelmointitapojen toimivuutta Unity-kehitysympäristössä. Opinnäytetyössä esitellään ohjelmoitua viitekehystä koodiesimerkein ja kerrotaan mitä vaikutuksia Unityn arkkitehtuurin ohittamisessa omalla viitekehyksellä on.

Opinnäytetyössä selviää, että uusien ohjelmointikonseptien omaksuminen on haastavaa ja niiden toteuttaminen käytännössä jo olemassa olevan arkkitehtuurin päällä vaikeaa. Oman viitekehyksen kehitys on kuitenkin suotavaa, sillä se parantaa koodin rakennetta, tekee koodipohjasta modulaarisemman ja toimii tehokkaammin, kuin pelkästään Unityn omien työkalujen käyttö. Opinnäytetyössä esiteltävää teoriaa ja koodiesimerkkejä on mahdollista käyttää oman viitekehyksen luomiseen Unityssä.

# ABSTRACT

**Author(s):** Laine Janne & Ylisiurua Juho

**Title of the Publication:** Data-Oriented Design and Functional Programming in Unity

**Degree Title:** Bachelor of Business Administration

**Keywords:** game programming, data-oriented design, functional programming, Unity, asynchronous programming, component model

The Unity development platform has become one of the most popular game engines used by both indie and AAA developers. However, Unity is hindered by its architectural choices and outdated design. The purpose of this bachelor's thesis was to research better ways to design game systems and to program in Unity. In addition, this thesis can act as a guide for readers, who are interested in alternative programming paradigms and methods to object-oriented programming.

The theoretical background consists of up-to-date information about data-oriented design and C#-specific functional programming. The theory is composed in a non-framework-specific way. The theoretical background was utilized to create an example game system in Unity.

The background consists of three sections. The first section discusses what is the status quo in game programming, including the special needs of game programming, a review of modern technology, the shortages of object-oriented programming and an overview of the Unity development platform. The second section examines data-oriented design and the third section explores functional programming in C#. The last section introduces examples on how to program a system for games in Unity using the previously discovered methods.

It was learned that adopting new programming concepts is challenging and applying them in practice can prove itself difficult. However, creating a framework for a specific game can be beneficial, as it increases the modularity of the code and the program's performance is improved. The theory and code examples introduced in this thesis can be used to create a framework for games in Unity.

SISÄLLYS

KÄYTETYT TERMIT JA LYHENTEET

| | |
|---|---|
| Abstrahointi | Asioiden määrittely tarpeellisten toimintojen ja ominaisuuksien kautta, ottamatta kantaa siihen, miten ne on toteutettu |
| Delegaatti | Metodiosoitin, eli delegaattia kutsuttaessa kutsutaankin delegaattia kuuntelevia metodeja. |
| Enumeraatio | Luokka, jonka kaikki ilmentymät, eli arvot, on määritelty valmiiksi. |
| Instanssi | Esiintymä, eli luokan edustaja ohjelmassa. (engl. instance) |
| Jätekuorma | Roskienkeruujärjestelmän toiminnasta syntyvä ylimääräinen muistinvaraus. (engl. overhead) |
| Kapselointi | Yhteen kuuluvien tietojen ja toimintojen kokoamista yhdeksi kokonaisuudeksi kutsutaan kapseloinniksi. |
| Kehys | Pelin aikana yksi staattinen ”kuva”, jonka aikana pelisilmukan kaikki toiminnallisuus suoritetaan. (engl. frame) |
| Kiintoarvo | Arvo, jota ei muuteta sen jälkeen, kun se on kerran alustettu. |
| MapReduce-ohjelmointimalli | Ohjelmointimalli, jota voi käyttää suurten tietomäärien käsittelyyn jakamalla laskentatyö usean tietokoneen kesken. |
| Ohjelmointiparadigma | Ohjelman/ohjelmointikielen taustalla oleva ajatus, kuinka ohjelmointitehtävän voi ratkaista. |
| Palveluliittymä | Kokonaisuus johon voidaan määritellä erilaisia funktioita, palveluita, parametreineen ja lisätä ne mihin tahansa luokkaan. (engl. interface) |

| | |
|---|---|
| Post-mortem | Jälkiselvittely, jossa projektin kulku, prosessit ja saavutettu lopputulos käydään läpi ohjelmistoprojektin lopussa. |
| Roskienkeruu | Säätelee muistin käyttöä automaattisesti. Varaa ja vapauttaa muistia ohjelmalle. (engl. garbage collection) |
| Sarjallistaminen | Muutetaan objektin tila muotoon, jossa se voidaan tallettaa tiedostoon, leikepöydälle tai siirtää verkon yli toiselle koneelle. (engl. serialization) |
| Säieturvallinen | Säikeet huomioivaa aliohjelmaa sanotaan säieturvalliseksi. (engl. thread safety) |
| Viitekehys | Ohjelmistotuote, joka muodostaa rungon päälle rakennettavalle tietokoneohjelmalle/pelille. (engl. software framework) |

# 1 JOHDANTO

Ohjelmointi on tiedon muokkaamista: Se on ohjeiden luomista koneelle, joka käyttää käyttäjän antamia syötteitä ja tulostaa uutta tietoa. Peli ei ole muuta kuin reaaliaikaisesti ja nopeasti annettuun syötteeseen reagoiva interaktiivinen ohjelma. Nykyisin yleisesti ohjelmoinnissa, ja pelien ohjelmoinnissa, käytetään olio-ohjelmointimallia. Olio-ohjelmointi on oman aikansa tuote, joka kehittyi, kun proseduurilliset kielet olivat liian hankalia ja työläitä käyttää yhä laajenevien kaupallisten sovelluksien kehittämiseen. Nykyisin laitteiden teknologian kehittyminen moniytimisiksi ja muistin hidas kehitys on tehnyt olio-ohjelmoinnista - yksiytimellisille laitteille optimoidusta mallista - liian tehottoman.

Työssä on tarkoituksena tutkia vaihtoehtoisia keinoja, dataorientoinutta suunnittelua ja funktionaalista ohjelmointia, olio-ohjelmoinnista aiheutuvien ongelmien ratkaisemiseen. Opinnäytetyössä käsitellään dataorientoitunutta suunnittelua ja funktionaalista ohjelmointia Unity-kehitysympäristön näkökulmasta. Työn tavoitteena on tarjota olio-ohjelmointiin perehtyneille ohjelmoijille vaihtoehtoinen tapa kehittää pelejä Unityllä, mutta soveltaen myös muilla kehitysympäristöillä. Oppimisnäkökulman kannalta asiat pyritään käsittelemään selkeästi ja kattavasti, sekä aiheista esitellään koodiesimerkkejä havainnollistamaan uusia ohjelmointimalleja.

Opinnäytetyö toteutetaan perehtymällä kirjallisuuteen ja Internet-lähteisiin, joista kerätään tietoperusta dataorientoituneeseen suunnitteluun ja funktionaaliseen ohjelmointiin. Dataorientoitunut suunnittelu on melko uusi ohjelmistojen suunnittelumenetelmä, eikä kirjallisuutta ole saatavilla, joten Internet-lähteisiin tukeuduttiin laajasti. Työssä käsitellään olennaiset menetelmien työkalut, kuten funktiotyypit, tietorakenteiden jäsentely, komponenttimalli ja ohjelmien kirjoitustapa.

Työssä tietoperusta sisältää perustiedot dataorientoituneesta suunnittelusta ja funktionaalisesta ohjelmoinnista, niiden hyödyistä verrattuna olio-ohjelmointiin ja koodiesimerkkejä menetelmien käyttämisestä C#-ohjelmointikielellä Unityssä. Lisäksi, opinnäytetyössä tarkastellaan olemassa olevia viitekehyksiä Unitylle, jotka perustuvat dataorientoituneeseen suunnitteluun ja funktionaaliseen ohjelmointiin.

Opinnäytetyön käytännön osuus toteutettiin tietoperustan pohjalta luomalla viitekehys Unity-kehitysympäristölle. Vaikka viitekehys on tehty Unityä varten, on se suunniteltu niin, että vaihtamalla tietorakenteiden tyypit voidaan sitä käyttää muissakin pelimoottoreissa. Viitekehyksessä esitellään yksi tapa tietorakenteiden ja pelilogiikan luomiselle Unityssä, ja koodiesimerkkejä havainnoimaan esiteltyä mallia. Lisäksi, viitekehyksestä on saatavilla esimerkkiprojekti Github-palvelusta. Luotua viitekehystä testattiin vertaamalla sitä kahteen muuhun malliin, logiikka-ajurilla toimivaan kokonaisuuteen ja MonoBehaviour-luokilla toteutettuun. Testaukset tehtiin Unityn Profiler-työkalua käyttäen.

# 2 PELINKEHITYKSEN NYKYTILANNE

Tietoteknisten laitteiden lisääntynyt teho ja kehittyneet arkkitehtuurit ovat edesauttaneet myös tietokoneohjelmistojen kehitystä. Yksi suurimmista menestystarinoista näiden ohjelmistotyyppien saralla ovat pelit, joiden kehitys niin grafiikassa kuin pelattavuudessakin on ollut silminnähtävä viimeisen kolmenkymmenen vuoden aikana. Uudistuva teknologia vaatii kuitenkin jatkuvaa uudistusta myös pelintekoprosessissa, jotta kasvavasta tehosta saataisiin jatkossakin kaikki mahdollinen hyöty irti.

## 2.1 Peliohjelmoinnin erityisvaatimukset

Pelinkehitys on yksi sovelluskehityksen alalajeista. Sovelluskehityksestä periytyminen luo pelinkehitykselle samoja tarpeita toteutuksen kannalta ja nämä tarpeet voidaankin jakaa seuraavaan kolmeen kohtaan:

1. Nopea suorituskyky.
2. Ominaisuuksien nopea toteutus.
3. Hyvä arkkitehtuuri.

Nopea suorituskyky ja ominaisuuksien nopea toteutus ovat molemmat tärkeitä asioita pelinkehitykselle. Valitettavasti nämä kaksi asiaa myös sotivat toisiaan vastaan, sillä nopeasti tehty asia ei usein toimi nopeasti. Pelinkehityksessä on tärkeää löytää näiden kahden välillä tasapaino jonka löytämistä edesauttaa hyvän arkkitehtuurin suunnittelu. Pelinkehityksessä pelin arkkitehtuurin pitääkin olla samaan aikaan joustava ja tukea uusien ominaisuuksien toteutusta, mutta myös nopeasti toimiva. (Nystrom 2014, 11 – 13)

Kun tavalliselle sovelluskehitykselle kriittisintä on luoda joustava ja hyvin skaalautuva arkkitehtuuri, on peleille usein tärkeämpää saada luotua hyvä ja tökkimätön pelikokemus. Pelille, joka on ohjelman käyttäjän, eli pelaajan, antamiin komentoihin reagoiva ohjelma, tärkeää on sujuvuus. Tämä vaatii sen, että pelin taustalla

pyörivä koodi on tehokas ja nopeasti toimiva. Tehokas koodi vaatii aina optimointia, joka vie usein paljon kehitysaikaa. Taatakseen sujuvan pelikokemuksen projektien loppuvaiheissa onkin hyvä keskittyä optimointiin, jotta pelaajat saisivat parhaan mahdollisen kokemuksen. Pelinkehityksessä nopeaa suorituskykyä tukee aina hyvin luotu arkkitehtuuri, jota on helppo optimoida esimerkiksi karsimalla siitä turhia ominaisuuksia. Toisaalta pitkälle optimoitu koodi tekee arkkitehtuurista vähemmän joustavan, mikä vaikeuttaa uuden luomista. (Nystrom 2014, 10 – 12)

Toinen tärkeä huomioitava seikka pelinkehitykselle on uuden tekemisen helpottaminen. Hauska ja viimeistelty pelikokemus saadaan iteroimalla ja kokeilemalla erilaisia asioita. Mitä nopeammin uusia ideoita ja ominaisuuksia voidaan kokeilla, sitä nopeammin saadaan selville toimivatko ne pelissä. Joustavasti toteutettu arkkitehtuuri mahdollistaa peli-ideoiden ja ominaisuuksien nopean testaamisen. Luomalla keskenään yhteensopivia komponentteja, voi kehittäjä koota niistä uusia asioista helposti. Unity-kehitystyökalu on esimerkki joustavasta arkkitehtuurista, sillä se ei rajoita merkittävästi minkälaisia pelejä pelimoottorilla voi tehdä. Lisäksi se helpottaa pelinkehitystä sisältämällä joukon valmiita komponentteja. Valitettavasti joustavalla arkkitehtuurilla toteutettu peli ei ole kovin hyvin optimoitu ja sen suurin heikkous onkin hitaus, joka on merkittävä haitta pelin sujuvuudelle. (Nystrom 2014, 11 – 13; Unity Technologies 2017)

Tasapainoilu hyvän suorituskyvyn ja nopeasti toteutettavien ominaisuuksien välillä voi olla haastavaa. Usein kompromissina koodipohjan voi pitää joustavana siihen asti, kunnes suunnittelu on viimeistelty ja tiedetään tarkalleen mitä peli sisältää. Tämän jälkeen koodia voidaan alkaa optimoida. Tällaisen kompromissin teko voi kuitenkin osoittautua tehottomaksi ratkaisuksi ja varsinkin projektin loppupuolella tehtävä optimointi voi viedä liikaa aikaa. Tämän takia projektin alussa kannattaa keskittyä hyvän arkkitehtuurin luomiseen, joka tukee molempia pelinkehityksen pääperiaatteita. Vaikka jokainen muutos uuden ominaisuuden myötä vaatii enemmän työtä, hyvä arkkitehtuuri mahdollistaa nopeamman kehitystyön pitkällä aikajaksolla minimoiden lopussa tarvittava optimoinnin tarpeen. (Nystrom 2014, 12 – 13)

Hyvän arkkitehtuurin luominen on kolmas, mutta ei vähäpätöisin pelinkehityksen päätarpeista, sillä se tukee kahta aiempaa. Hyvin tehty arkkitehtuuri sallii pitkäjänteisen projektinkehityksen niin tavallisessa sovelluskehityksessä, kuin pelinkehityksessäkin. Hyvä arkkitehtuuri helpottaa koodin ymmärtämistä ja tekee koodista helpommin muokattavan. Se myös edesauttaa, että ohjelmaan tehtävät muutokset eivät riko mitään olemassa olevaa. (Nystrom 2014, 9 – 10)

Hyvän suunnittelun mitta on se, että kuinka helposti muutoksia voidaan tehdä ohjelmaan. Muutosten tekemistä helpottaa eri koodien toisistaan eristäminen. Eristäminen tarkoittaa, että toiminnot jaetaan niin pieniksi osiksi, että minimoidaan tarvittava tieto ja ymmärrys mikä toiminnosta pitää olla. Lisäksi osien tulisi tietää mahdollisimman vähän toisista koodiosista. Mitä pienempiä osat ovat, sitä helpompi muutoksia on tehdä niihin myöhemmin. Lisäksi osaan, joka ei vaikuta toisiin osiin, on helppo tehdä muutos ilman, että se rikkoo mitään toista koodin osaa. Pieniin osiin jaettu ja selkeästi kirjoitettua koodia on helppo optimoida, joten se tukee nopeaa suorituskykyä. Myös uusien ominaisuuksien tuominen peliin on helppoa, kun uusi koodi ei riko vanhaa. (Nystrom 2014, 10 – 13)

Toimintojen jakaminen ja koodin abstrahointi ovat hyvän pelisysteemisuunnittelun peruskiviä. Aina kun koodia jaetaan tai abstrahoidaan, tehdään valistunut arvaus minkälaista järjestelmää ja joustavuutta tulevaisuudessa tarvitaan. Koodin lisääminen ja systeemin kompleksisuus lisäävät aikaa, joka täytyy käyttää koodin kehittämiseen, virheenkorjaukseen ja ylläpitämiseen. Jos ohjelmoitu järjestelmä vastaa tarpeita, eli arvaus osui oikeaan, säästää aiemmin käytetty vaiva aikaa myöhemmin. Pelin systeemiä ja kehitystyökaluja ohjelmoitaessa kannattaa aina pitää mielessä minkä takia niitä ohjelmoidaan. Hienoa arkkitehtuuria suunniteltaessa ja ohjelmoitaessa voi päätarkoitus, eli peli, unohtua. Suoraviivaista 2D-tasoloikkapeliä varten on turha luoda monimutkaisia kehitystyökaluja, jotka sopisivat paremmin MMORPG-pelin luomiselle. (Nystrom 2014, 11 – 13)

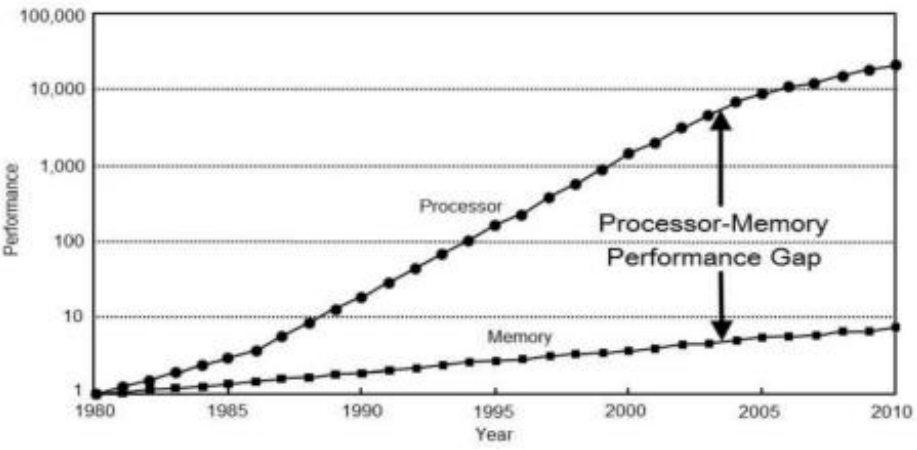
## 2.2 Nykyteknologian uudet haasteet pelinkehitykselle

Erityisesti viime vuosina pelikäyttöön soveltuvien laitteiden määrä on kasvanut räjähdysmäisesti. Suurin tekijä laitepohjan kasvussa on ollut pelikäyttöön soveltuvien mobiililaitteiden, kuten älypuhelimien ja tablettien, yleistyminen kotitalouksissa. Sen lisäksi, että laitteiston laskentateho on kasvattanut pelientekoa, sitä on helpottaneet myös ilmaisiksi muuttuneet pelimoottorit, näistä esimerkkinä Unity, jolla tehdään suuri osa nykypeleistä. Pelinkehityksen helpottumisen näkee muun muassa Sergey Galyonkinin vuonna 2016 tekemästä pelien digitaalijakelijaan Steamiin liittyvästä tutkimuksesta: Steamissä myynnissä olevista peleistä jopa lähes 40% on kehitetty vuonna 2016 – ja jopa 80 % oli kehitetty 2014 tai sen jälkeen. (Unity Technologies 2017)

Unityssä voi ohjelmoida C#:lla ja Javascriptillä, joilla yksi käytetyimmistä ohjelmointitavoista on olio-ohjelmointi (OOP) (Unity Technologies 2017). Se on yksi käytetyimmistä ohjelmointityyleistä sen helpon opittavuuden vuoksi. Olio-ohjelmoinnin 30 vuoden olemassaolon aikana teknologia on kuitenkin mennyt eteenpäin, eikä tätä yksittäistä prosessoria varten optimoitua koodaustapaa ole varsinkaan mobiilialustapeleille enää järkevä käyttää. Kehittäjät ovat reagoineet teknologian kehitykseen ja olio-ohjelmoinnin rinnalle onkin kehitetty muita ohjelmointitapoja, joita voi käyttää hyödyksi pelikehityksessä. Nykyaikana esiintyviä ongelmia, joihin vanha OOP ei osaa vastata niin hyvin kuin uudet tavat, aiheuttavat esimerkiksi muistin hidas kehittyminen verrattuna prosessoreihin, sekä suorittimien moniytimisiksi muuttuminen. (Fabian 2013; Milewski 2012)

Muistin, ja varsinkin välimuistin, hidas kehittyminen on aiheuttanut sen, että muistista on tullut laitteistojen pullonkaula, joka on hyvä huomioida pelinkehityksessä. Muistin kehityksen hitautta havainnollistetaan kuvassa 1. Vaikka C#:ssa on automaattinen roskienkeruu, muisti on silti hyvä huomioida ohjelmoitaessa. C#:n käyttämättä .NET-ympäristössä voi esiintyä muistivuotoja, jos uusia objekteja, joihin viitataan muissa objekteissa, luodaan jatkuvasti. Lisäksi, huono suunnittelu voi johtaa epätehokkaaseen muistin käyttöön. Esimerkiksi lukuisien kopioiden luominen string-muuttujasta ei ole tehokasta. Edellä mainittujen huomioonotettavien

asioiden lisäksi C#:n virtuaalikoneen automaattien roskienkeruu voi hidastaa ohjelman toimimista, mikäli se joutuu koko ajan varaamaan ja vapauttamaan muistia. (Fabian 2013; DICE 2010; Totin 2016, 3-4)




Kuva 1. Prosessorien ja muistien tehon kehitys (DICE 2010)

Moniytimiset suorittimet ovat 2000-luvun kehityksen suunta, joka on mahdollistanut prosessointitehon jatkuvat kasvun. Tällaiset suorittimet mahdollistavat sen, että ohjelman eri tehtävät voi jakaa eri ytimille, jotka suorittavat ne rinnakkaisesti samaan aikaan, ja näin nopeuttavat ohjelman toimintaa. Vaikka tämä kehityksen suunta on ollut erittäin hyödyllinen, on se luonut ongelmia ohjelmien luojille: rinnakkainen ohjelmointi on hyvin haastavaa. Monien ohjelmien tehtävät on hyvin vaikea jakaa pieniksi osiksi, joita eri ytimet voisivat samanaikaisesti suorittaa. Tehtävää ei helpota se, että käytetyimmät imperatiiviset ohjelmointimenetelmät ovat optimoitu yksiytimiselle suorittimille suunnatulle peräkkäisohjelmoinnille, eivätkä näin taivu kovin helposti rinnakkaisohjelmointiin. (Suleman 2011; Milewski 2012)

Suurin haaste rinnakkaisessa ohjelmoinnissa on riippuvuuksien hallinta. Kaikki ohjelman suorittamat tehtävät eivät aina ole täysin erotettavissa toisistaan, vaan vaativat toisinaan, että toinen tehtävä on suoritettu loppuun ennen oman suoritusta. Rinnakkaisessa ohjelmoinnissa on huolehdittava, että tällaisia riippuvuuksia on mahdollisimman vähän, sillä niiden olemassaolo hidastaa ohjelman suoritusta. Hidastuminen johtuu siitä, että ytimet joutuvat odottamaan aika ajoin jonkin toisen ytimen ajossa olevaa tehtävää. Siksi onkin järkevää suunnitelma ohjelma ja sen tietorakenteet kokonaisuuksiksi, jotka eivät ole riippuvaisia tai jaettuja keskenään

ja antaa nämä kokonaisuudet eri säikeiden suoritettavaksi. Suurimmat ongelmat riippuvuuksien esiintyessä ovat kilpailutilanne ja lukkiutuminen. (Suleman 2011)

Varsinkin imperatiivinen ohjelmointi, jonka yksi perusperiaatteista on muuttujien muokattavuus, on haavoittuvainen kilpailutilanneilmiölle. Kilpailutilanne syntyy, kun kaksi tai useampi säie yrittää samanaikaisesti muuttaa samaa jaettua tietorakennetta. Jos muuttujaa muokataan samanaikaisesti kaikki muutokset, joita siihen tehdään eivät säily. Pelkästään operaatiot, jotka viimeisenä muuttujaan kirjoittanut säie teki, säilyvät ja muut muutokset pyyhitään pois ylikirjoituksen yhteydessä. Tätä ilmiötä on havainnoitu kuvassa 2, jossa lopputulos on 20 tai 17 riippuen siitä, kumpi säie tallentaa arvon viimeisenä. Tämä toiminta on hyvin ongelmallista, sillä tietoa katoaa, eikä haluttu tietorakenne ole halutussa kunnossa kaikille säikeille, jotka käyttävät sitä. Tällaisen ongelman paikannus on myös hyvin haastavaa, sillä sen esiintyminen ei ole jatkuvaa. (Milewski 2012; Microsoft Corporation 2012; Leveson & Turner 1993, 18-41)

```
int value = 15;
//Säie 1
value += 5;
//Säie 2
value += 2;
```

Kuva 2. Mahdollinen kilpailutilanne

Kilpailutilanne ilmenee vain, jos useampi säie yrittää muokata samaa tietorakennetta samanaikaisesti. Tämä voi olla hyvin harvinaista ohjelman ajossa, mutta vaarallista niin toteutuessaan. Samanaikaisen muokkaamisen luominen keinotekoisesti vianetsintä tarkoituksessa on hyvin haastavaa ja kilpailutilannemahdollisuuksia jää satunnaisesti huomaamatta. Esimerkiksi vuosien 1985 ja 1987 välillä sattui muutama kuolemantapaus Therac-25 sädehoitolaitteen vuoksi, sen antaessa satunnaisesti potilaille kuolettavan määrän säteilyä. Pääsyy onnettomuuksien taustalla olivat laitteen ohjelmointivirheet, jotka aiheuttivat kilpailutilanteen, jossa säteen määrä kasvoi ihmiselle vaarallisen suureksi. (Milewski 2012; Microsoft Corporation 2012; Leveson & Turner 1993, 18-41)

Yksi tapa estää rinnakkaisessa ohjelmoinnissa esiintyviä kilpailutilanteita on lukita muokattava tietorakenne. Kun säie tahtoo muokata muuttujaa, se voi samalla lukita muuttujan muokkaamisen itselleen. Kun toinen säie tahtoo samanaikaisesti

muokata samaa muuttujaa, joutuu se ensin odottamaan, että ensimmäinen säie on tehnyt muutoksensa muuttujalle. Kun tarvittavat muutokset muuttujalle on tehty voi ensimmäinen säie avata lukituksen, jonka jälkeen toinen säie pääsee muokkaamaan muuttujaa. Lukitsemismenetelmässä on kaksi ongelmaa, joista toinen on itse konseptissa. Muuttujan lukitseminen pakottaa toisen säikeen odottamaan, eli toisin sanoen suorittamaan tyhjää. Odotteluun käytetty aika on aina hukkaan heitettyä aikaa. Onkin järkevämpää suunnitella ohjelman osat niin, että lukitsemista ei tarvita, eli, että kilpailutilanteita ei synny. (Microsoft Corporation 2012; Suleman 2011)

Lukitsemisen toinen ongelma, on sen synnyttämä ilmiö – lukkiutuminen. Se syntyy, kun kaksi säiettä lukitsee eri muuttujan samaan aikaan ja yrittää sen jälkeen lukita toisen muuttujan, jonka toinen säie juuri oli lukinnut. Tämä johtaa siihen, että molemmat säikeet jäävät odottamaan, että toinen säie vapauttaisi tietorakenteen, jota se itse tarvitsisi omiin operaatioihinsa. Tuloksena on molempien säikeiden toimimattomuus, joka ei lakkaa ennen ohjelman ajon lopettamista. Tätä ongelmaa on havainnoitu kuvassa 3. Kuten kilpailutilanteen, myös mahdollisen lukkiutumisen paikantaminen voi olla hyvin haastavaa, sillä virhe ei tapahdu läheskään aina suorituksen aikana. Se vaatii aina kahden eri säikeen yhtäaikaisen eri muuttujan lukitsemisen. (Microsoft Corporation 2012; Suleman 2011)

```
//Säie 1
void Function1(){
     lock (object1) {
          lock (object2) {
               //Suorita operaatiot object1- ja object2-muuttujille,
               jotka vaativat tiedon lukemista ja kirjoittamista.
          }
     }
}
//Säie 2
void Function2(){
     lock (object2) {
          lock (object1) {
               //Suorita operaatiot object2- ja object1-muuttujille,
               jotka vaativat tiedon lukemista ja kirjoittamista.
          }
     }
}
```

Kuva 3. Tilanne, jossa kaksi säiettä lukitsevat eri muuttujat, joista molemmat ovat riippuvaisia aiheuttaen lukkiutumisen

## 2.3 Olio-ohjelmointi ja periytyminen

Ennen olio-ohjelmointia nousemista käytetyimmäksi ohjelmointityyliksi, suuri osa yrityskäyttöön tehdyistä ohjelmista tehtiin proseduurillisilla kielillä, kuten C, Pascal ja Fortran. Kun ohjelmien koko kasvoi, proseduurillisella tavalla ohjelmoiduista ohjelmista alkoi kuitenkin tulla liian monimutkaisia hallita ja ohjelmointivirheiden korjaamisesta tuli hyvin vaikeaa. Tämän vuoksi kehitettiin strukturoitu ohjelmointitapa, joka mahdollisti ohjelmien jakamisen osiin – funktioihin ja proseduureihin. Tämä ei kuitenkaan ratkaissut kaikkia ongelmia, vaan esiin alkoi tulla uusia strukturoituun ohjelmointitapaan liittyviä ongelmia:

- Ohjelmat olivat edelleen hankalia hallita.
- Toimintojen muokkaaminen oli vaikeaa ilman, että muokkaus ei olisi muuttanut muun ohjelman toimintaa.
- Uusia ohjelmia oli vaikea ohjelmoida tyhjästä.
- Ohjelmoijien täytyi tietää miten jokainen ohjelman osa-alue toimi ja he eivät pystyneet rajoittamaan työskentelyään vain yhteen osaan ohjelmaa.
- Liiketoimintamallien muokkaaminen ohjelmointimalleiksi oli vaikeaa.
- Proseduurilliset ohjelmat toimivat hyvin yksinään, mutta ne eivät integroituneet hyvin muihin systeemeihin. (Clark, 2011, 2)

Edellä mainittujen heikkouksien lisäksi tietokoneiden kehittyminen toi lisää haasteita strukturoidulle ohjelmointitavalle, kuten:

- Muut kuin ohjelmoijat vaativat suoran pääsyn ohjelmiin graafisten käyttöliittymien kautta heidän pöytäkoneiltaan.
- Ohjelmien käyttäjät vaativat intuitiivisempaa ja vähemmän strukturoitua käyttökokemusta.
- Tietokoneet kehittyivät malliin, missä liiketoimintalogiikka, käyttöliittymä ja ”back end”-tietokanta oli väljästi yhdistetty omina osinaan, ja joihin päästiin käsiksi Internetin kautta. (Clark, 2011, 3)

Edellä mainittujen asioiden johdosta ohjelmistokehittäjät kääntyivät olio-ohjelmointityylien ja -kielien puoleen. Olio-ohjelmointi mahdollisti intuitiivisemman siirtymi-

sen liiketoiminnan analyysimalleista ohjelmiston toteutusmalleihin. OOP-ohjelmointi mahdollisti myös muutoksien nopeamman ja tehokkaamman kehityksen, joita edesauttoi tiimityöskentelymahdollisuus, jossa ohjelmoija pystyi kehittämään vain yhtä osaa systeemistä sekä koodin osien mahdollinen uusiokäyttö. Lisäksi, olio-ohjelmointi toi paremman integraation väljästi yhdistettyjen jaettujen tietokonesysteemien kesken, sekä kehittyneen integraation moderneihin käyttöjärjestelmiin. Myös graafisten käyttöliittymien kehittäminen parani OOP-ohjelmoinnin myötä. (Clark, 2011, 3)

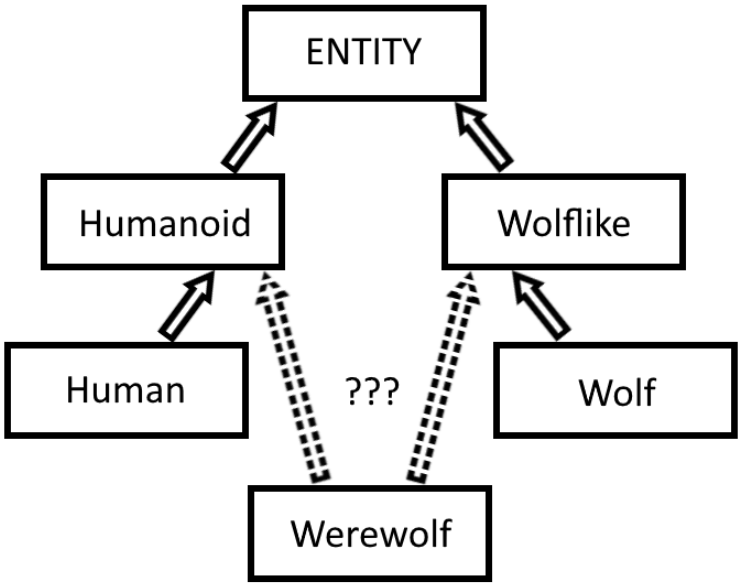
Olio-ohjelmoinnin konseptit alkoivat muotoutua 60-luvun puolivälissä yhdessä Simula-ohjelmointikielen kanssa ja kehittyä lisää 70-luvulla Smalltalk-ohjelmointikielen syntymisen myötä. Suuremman suosion olio-ohjelmointi saavutti kuitenkin vasta 80-luvun puolivälissä. Suosio kasvoi C++- ja Eifle-kielten yleistymisen johdosta. OOP jatkoi kasvuaan 90-luvulla, pääasiassa Javan, ja sen yleistymisen ansiosta. 2000-luvulla suosion kasvaminen jatkui, kun Microsoft julkaisi yhdessä .NET Frameworkin kanssa uuden OOP-kielen – C#:in. Sen lisäksi että olio-ohjelmointi tarjosi ratkaisun proseduraalisien ohjelmointitapojen ongelmiin, sen suosiota lisäsi se, että olio-ohjelmointi oli helposti opeteltavissa oleva tapa ohjelmoida, koska se oli helposti mallinnettavissa reaalimaailmaan. (Clark, 2011, 2; Sorva 2016)

Olio-ohjelmoinnissa reaalimaailmasta siirretään ajatus koodiin, jossa olio-ohjelmoinnin oliot ovat ikään kuin inhimillisiä toimijoita. Olioiden lisäksi metodit ovat olio-ohjelmoinnin yksi tunnuspiirteistä. Metodi on ohjelmassa oleva aliohjelma, jota käytetään tietorakenteiden muokkaamiseen. Helpon ymmärrettävyyden lisäksi olio-ohjelmoinnin vahvuuksiin kuuluu helppolukuinen koodirakenne. Helppo lukuisuus ja helppo ymmärrettävyys ovat syitä, miksi OOP-paradigma opetetaan usein ensimmäisenä tapana ohjelmoida ohjelmoijille. (Sorva 2016; Atehwa 2013; Porter 2012.)

Olioiden toteutus voidaan jakaa kahteen yleiseen kategoriaan: Periminen ja komponenttipohjaisuus. Olio-ohjelmoinnissa periminen on yleisemmin käytössä oleva toteutustapa. Perimisen avulla koodia voi erikoistaa ja yleistää. Perittävää luokkaa kutsutaan yli- tai isäntäluokaksi ja perivää luokkaa aliluokaksi. Perinnässä on yksi pääsääntö: Aliluokka perii yliluokan kaikki ominaisuudet. Lisäksi ohjenuora on, että

aliluokka toteuttaa vähintään yhden uuden ominaisuuden, jota ei ole yliluokassa. (Bennet, McRobb ja Farmer 2006, 78; Porter 2012.)

Perintämalli ei ole täysin ongelmaton tapa toteuttaa olioita. Projektin edetessä on tärkeää suunnitella luokkahierarkia niin, että mahdollisimman vähän koodia tarvitsee kirjoittaa uusille toteutettaville olioille. Olioiden monipuolistuessa kuitenkin myös luokkahierarkia monimutkaistuu, mikä johtaa siihen, että uusien olioiden luonti muuttuu vaikeammaksi. Tämä korostuu, kun uusi olio tarvitsee monipuolisesti ominaisuuksia hierarkian eri olioilta. Kuvassa 4. on havainnollistettu tällaista hierarkiaongelmaa. (Boreal Games 2013)



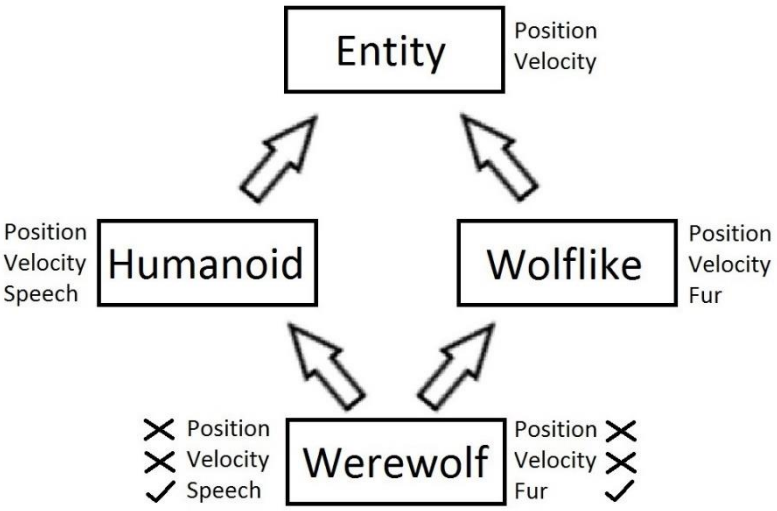

Kuva 4. Uuden olion sijoittamisvaikeus perintähierarkiassa

Yllä olevassa kuvassa on esitelty oliot Human ja Wolf, jotka jakavat joitain samankaltaisia ominaisuuksia. Tämän takia on järkevää sijoittaa edellä mainitut oliot samaan luokkahierarkiaan, joten niille on luotu yläluokka Entity. Esimerkkitapauksen peliin on haluttu lisätä myös muita sekä ihmisen, että suden kaltaisia olioita, joten entiteetin ja sen lapsiolioiden väliin on lisätty luokat Humanoid, mistä ihminen ja muut vastaavat voivat jatkossa periytyä, sekä WolfLike, josta kaikki suden kaltaiset oliot voivat periytyä. Ongelma syntyy, kun peliin halutaan lisätä luokka Werewolf, joka tarvitsee toimiakseen ominaisuuksia sekä Humanoid-luokasta, että Wolflike luokasta. Ilman, että Werewolf periytyisi molemmista luokista, ei voida välttää, että

jommankumman yläluokan ominaisuuksia uudelleen kirjoitettaisiin Werewolf lapsiluokkaan. Tämä lisää koodin päällekkäisyyttä ja tekee luokkahierarkian hyvin epäselväksi.

Perintä nitoo isäntäluokan ja sen aliluokat tiukasti yhteen. Yliluokan poisto tai muuttaminen yleensä rikkovat aliluokkien toiminnan, sillä aliluokat ovat kokonaan riippuvaisia yliluokastaan. Perintä on mekanismina jäykkä, sillä yliluokassa määritellyt metodit ja toiminnallisuus voivat olla taakkoja perivissä luokissa. Lisäksi muutosten tekeminen hierarkiaan voi olla projektin edetessä hankalaa. Aikaisin tehdyt päätökset voivat olla kohtalokkaita virheitä myöhemmin projektissa. Lisäksi jos moniperintää yläluokista ei käytetä, täytyy aliluokkiin usein kirjoittaa samaa tietoa ja ominaisuuksia, jota muista luokista jo löytyy, jotta ne saisivat useamman luokan toiminnallisuudet. Tämä teettää ylimääräistä työtä, kasvattaa ohjelman kokoa ja hidastaa sen toimintaa. (The Saylor Foundation 2010)

Vaikka moniperinnällä voitaisiin helposti korjata kuvassa 4 esiintyvä uuden olion luomisongelma, ei se kuitenkaan ole täydellinen ratkaisu olio-ohjelmoinnin perintäongelmiin. Moniperinnän suurin ongelma on niin kutsuttu timanttiongelma. Timanttiongelmassa kaksi luokkaa, kutsuttakoon niitä Humanoid- ja Wolflike-luokiksi, periytyvät samasta luokasta Entity. Jotta molemmista luokista saataisiin uuteen Werewolf luokkaan halutut ominaisuudet, laitetaan se periytymään moniperinnällä Humanoid- ja Wolflike-luokasta. Koska Humanoid ja Wolflike periytyvät Entity-luokasta, on niillä molemmilla oman kopionsa Entity-luokan metodeista ja muuttujista. Tämän myötä Werewolf-luokkaan periytyy kaksi kopiota Entity-luokan metodeista ja muuttujista. Tämä tekee kloonautuneiden ominaisuuksien kutsumisesta käytännössä mahdotonta. Timanttiongelmaa havainnollistetaan seuraavassa kuvassa.




Kuva 5. Moniperinnän timanttiongelma

Timanttiongelman lisäksi moniperintä aiheuttaa vikaherkkyyttä hierarkiassa ja lisää työtä sen suunnitteluvaiheessa. Moniperintä voi myös aiheuttaa pitkällä aikajaksolla muita ennalta-arvaamattomia ongelmia. Kaikki nämä ongelmat ovat vaikuttaneet siihen, että monet uudet kielet eivät ole mahdollistaneet moniperinnän käyttöä, mukaan lukien C#-kieli. (ProgrammerInterview 2016; Brumme 2004)

Olio-ohjelmointi mallintaa oikeaa maailmaa. Tämä lisää tarvittavan koodin määrää, sillä siirros reaalimaailmasta konekieleen tarvitsee ylimääräisen tason toimiakseen toisin kuin monissa muissa koneen käytänteitä paremmin huomioivissa ohjelmointitavoissa. Yksi olio-ohjelmoinnin haitoista onkin ohjelmien koko, joka johtuu tästä ylimääräisestä tarvittavasta koodista. Olio-ohjelmoinnin periaatteita noudattavien ohjelmien rakenne myös tyypillisesti vaatii enemmän komentoja toimiakseen. Ylimääräisen komennot hidastavat ohjelman toimimista. Koska pelien täytyisi interaktiivisuutensa vuoksi käsitellä tietoa nopeasti, voi olio-ohjelmoinnin hitaus olla ongelma etenkin peleissä. (The Saylor Foundation 2010)

Yksi olio-ohjelmoinnin perusperiaatteista oleva kapselointi ei sovellu erityisen hyvin nykyaikana pelien tekoon, sillä monissa projekteissa on tärkeää, että tieto on helposti muokattavissa myös muille kuin ohjelmoijille, mielellään sille osoitetussa käyttöliittymässä. Esimerkiksi Unity-pelimoottorin perusperiaatteisiin kuuluu julkisten arvojen muuttaminen editorinäkymässä, joten tällaiseen ympäristöön olio-ohjelmointi ei sovellu hyvin. (Unity Technologies 2017)

## 2.4 Unity

Unity on Unity Technologiesin kehittämä ja julkaisema pelimoottori. Sillä voi kehittää 2D- ja 3D-pelejä lähes jokaiselle nykyiselle pelialustalle. Unityn lähdekoodi on puoleksi suljettu. Se on muokattavissa vain maksullisilla Pro- ja Enterprise-lisensseillä. Ohjelmointikielistä Unity tukee C#:a ja JavaScriptiä. Unityn joustavuus kehitettäessä eri alustoille ja helppokäyttöinen käyttöliittymäeditori tekevät Unitystä suosituimman yksittäisen pelimoottorin. Vuonna 2016 peleistä 34% oli kehitetty Unityllä. Etenkin mobiilialustoille julkaistut pelit, jotka oli kehitetty Unityllä, olivat määrältään selkeästi muita edellä. (Unity Technologies 2017)

Unityn arkkitehtuuri perustuu olio-ohjelmointimalliin, jossa perinnän sijalta hyödynnetään komponenttimallia. Tässä Unityn perusperiaatteessa kaikki ominaisuudet jaetaan eri luokkiin jotka voi yhdistää Unity-peliobjekteihin. Luokat periytyvät MonoBehaviour-pääluokasta, joka mahdollistaa komponenttiluokkien yhdistämisen peliobjekteihin. Unity-peliobjekteihin voi lisätä myös itsetehtyjä komponenttiluokkia. Jokainen peliobjekti on varustettu Transform-komponentilla, joka sisältää objektin sijainnin, rotaation ja koon. Jokainen julkinen arvo itse kirjoitetuissa komponenttiluokissa, kuten myös Transform-komponentissakin, on muokattavissa Unity-editorissa. (Unity Technologies 2017, Mandalà 2015)

Unity on melko iäkäs moottori ja sen arkkitehtuuriin on jäänyt vanhentuneita ratkaisuja, jotka osittain johtuvat käytetyistä OOP-periaatteista. Unity ei esimerkiksi tue logiikan ja tietorakenteiden erottelua omiksi kokonaisuuksikseen - ilman kolmannen osapuolen viitekehystä. Tämä vähentää Unityllä tehdyn ohjelmiston modulaarisuutta huomattavasti, sillä komponenteista on tehtävä suuria, että ne kattaisivat yhden tehtävän toteuttamisen. Suuria kokonaisuuksia hallitsevat komponentit tekevät ja tietävät liikaa ja niiden uusiokäyttö on vaikeaa. Jotta komponentteja voisi uusiokäyttää helpommin, voidaan ne jakaa pienemmiksi kokonaisuuksiksi. Ongelma tällaisessa ratkaisussa kuitenkin on, että komponentit joutuvat vaihtelemaan toistensa kanssa paljon tapahtumia ja viestejä, eivätkä ne näin voi toimia täysin itsenäisesti. (Mandalà 2015, Oieng 2015)

Unityn yksi suurimmista ongelmista on ylimääräinen monimutkaisuus, joka tarvitaan kahden, toisistaan tietämättömien, MonoBehaviour-luokkien väliseen keskusteluun. Unityssä on muutama mahdollinen tapa edellä mainittujen luokkien väliseen yhteydenpitoon. Ensimmäinen tapa on, että saadakseen keskusteluyhteyden toiseen logiikkaan, koodissa voidaan käyttää viittaushakua, esimerkiksi ”GameObject.Find” tai muuta vastaavaa funktiota, jolla tarvittavat viittaukset saadaan haettua. Tämän jälkeen logiikka voi kutsua toisen logiikan metodeita. Etsimismetodien käyttö on kuitenkin hyvin hidasta ja tätä tulisi välttää varsinkin ajon aikana. Lisäksi se on hyvin virhealtista, sillä objektin nimen muuttaminen rikkoo viittaukset siihen. (Mandalà 2012; Mandalà 2015)

Toinen vaihtoehto luokkien väliseen keskusteluun on, että objekti, jota halutaan kutsua, on Singleton. Singleton on luokka, josta on olemassa vain yksi instanssi, ja siihen on globaali pääsy kaikista luokista (Nystrom, 2014. 59). Singletonien käyttäminen rikkoo kapseloinnin, se piilottaa riippuvaisuuksia ja niiden yksikkötestaus on mahdotonta. Singletoneihin kirjoitettujen tilojen hallitseminen ja tarkkailu on myös hyvin vaikeaa. Vaikka Singletonit voivat tuoda hyötyä projektille alussa, sen ongelmat korostuvat sitä myötä mitä suuremmaksi projekti kasvaa. (Mandalà 2012)

Viimeinen, ja huonoin tapa MonoBehaviour-luokkien väliseen keskusteluun on Unityn SendMessage-komento. Komennon idea sinällään on oivaltava ja erillisen tapahtumanhallintaluokan kaltainen hyvä ratkaisu, mutta toteutus ontuu. ”SendMessage” lähettää kutsun ajaa halutun niminen metodi kaikille MonoBehaviour-luokan periville luokille peliobjektissa. Ensimmäinen vika kutsussa on, että se on jopa kaksi kertaa hitaampi kuin suora funktiokutsu. Lisäksi ”SendMessage” voi lähettää vain yhden parametrin metodille. Sen käyttö on myös hyvin riskialtista, sillä jos kutsuttavan metodin nimeä muutetaan, SendMessage-komento ei tavoita oikeaa metodia eikä varoita ohjelmoijaa kääntäjän virheilmoituksella. Tällaisia ohjelmointivirheitä on hyvin vaikea etsiä. Viimeinen vika SendMessage-komennossa on, että viestejä ei voi lähettää MonoBehavioureille, jotka ovat muissa peliobjekteissa. (Mandalà 2013; Sweeney 2014)

Kun logiikkakomponentti luodaan, on sen pakko periytyä MonoBehaviour-luokasta, jotta sitä voidaan käyttää peliobjekteissa. Tämä tuo koodiin paljon ominaisuuksia, joita ei välttämättä tarvita, joka taas luo turhaa rasitusta suorituskyvylle. Varsinkin monen MonoBehaviourin perivän luokan luominen, jotka toteuttavat Unityn Update-metodin, aiheuttaa selkeästi suuremman haitan suorituskyvylle kuin erillisen Update-hallintaluokan käyttäminen (Simonov 2015). Unityn päivitysfunktiot sisältävät taustalla paljon virheensietokykyyn liittyviä ominaisuuksia, jotka luovat paljon ylimääristä jätekuormaa prosessorille prosessoitavaksi. Täten monen itseään päivittävän logiikan luominen ei ole Unityssä kannattavaa. (Mandalà 2012)

Transform-komponentin myötä jokainen peliobjekti on sijoitettu pelimaailmaan. Sen olemassaolo tekee peliobjektihierarkian haasteelliseksi, sillä jäsenneltyä, usean objektin sisältävää hierarkiaa ei ole järkevä tehdä. Tämä johtuu siitä, että jokainen Transform-komponentti lisää ylimääräisen matriisikertolaskun ajettavaksi joka kehyksessä. Unity ikään kuin laskee jokaiseen peliobjektiin grafiikkaa näytettäväksi, sillä muuten sijainnista, rotaatiosta ja koosta ei olisi mitään hyötyä. (Mandalà 2012)

# 3 DATAORIENTOITUNUT SUUNNITTELU

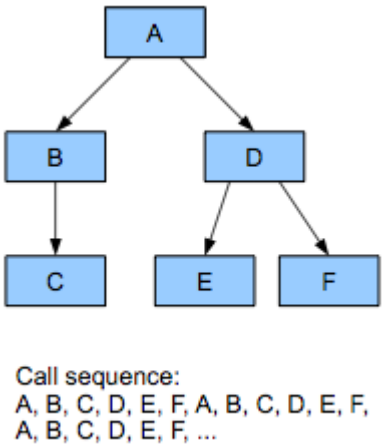
Dataorientoitunut suunnittelu (DOD) on ohjelmointiparadigma, jonka juuret yltävät vuosikymmeniä taaksepäin, mutta joka alkoi yleistyä vasta Noel Llopisin artikkelin myötä vuonna 2009. Suurin syy DOD:in taustalla pysymiseen oli 80-luvulla alkanut olio-ohjelmoinnin räjähtävä suosio. Llopisin artikkelin jälkeen dataorientoitunut suunnittelu on noussut nopeasti monien pelifirmojen arkkitehtuuriseksi pohjaksi. Esimerkiksi ruotsalainen videopelejä kehittävä DICE, sekä enemmän mobiilipelien kehittämiseen keskittyvä Wooga, ovat ottaneet käyttöön dataorientoituneen suunnittelun perusteet arkkitehtuurisissa ratkaisuissaan. DOD-malli on tarjonnut ratkaisun nykyteknologialle syntyneisiin pulmiin, joihin imperatiiviseen paradigmaan kuuluva olio-ohjelmointi ei ole pystynyt kunnolla vastaamaan ja kehittämään itseään. Dataorientoitunut suunnittelu tarjoaa ratkaisun muun muassa prosessorien ja muistien välisistä suurista nopeuseroista johtuviin ongelmiin, sekä se helpottaa nykyaikana yleisille moniydinprosessoreille kehittämistä. (Llopis 2009; Fabian 2013)

## 3.1 Lähtökohdat

Dataorientoituneen suunnittelun tärkein lähtökohta on tehostaa prosessoreita hitaamman muistin käyttöä. Tärkein työkalu tähän on parantaa muistiin tallennettuja tietorakenteita käsittelyä. Dataorientoituneessa suunnittelussa ajattelumalli ohjelman toteutukseen onkin päinvastainen verrattuna perinteisempään olio-ohjelmointiin. Olio-ohjelmoinnin toteutuksessa ohjelmoija pyrkii tuottamaan mahdollisimman järkevästi kirjoitettua ja luettavaa koodia, kun taas muistinhallinta ja tiedon siirtäminen ovat ikäviä tehtäviä, joiden toteuttaminen on toissijaista. Vastaavasti DOD-malli siirtää perspektiivin tietorakenteiden ja muistin hallintaan. Koodi on vain näiden tehtävien suorittamiseen käytettävä työkalu. Syy perspektiivin vaihtoon on se, että nykyajan laitteistolla ei ole väliä kuinka tehokkaita ja luettavia algoritmeja kirjoitetaan, jos ne käsittelevät tietoa epätehokkaasti. Tehokkaasti prosessorille op-

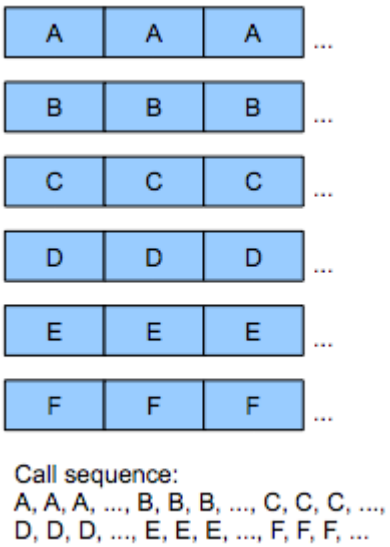
timoidulla koodilla ei ole merkitystä, kun prosessori joka tapauksessa joutuu odottamaan tietorakenteiden käsittelyä hoitavia toimintoja muistissa. (Mitton 2015; Llopis 2009).

Olio-ohjelmoinnin erilaiset periaatteet perustuvat perintä-, hallinta- ja viestinlähetyspuihin, joten myös tietorakenteet on järjestelty puumaisesti. Tämä johtaa siihen, että kun oliolle suoritetaan operaatioita, kutsuu se usein muita olioita alempana puussa. Tämä vaatii ylimääräistä tiedon muuttamista oikeanlaiseksi, joka on hidasta. Olioryhmien, jotka suorittavat samoja operaatioita, iteroiminen johtaa tapahtumasarjojen suorittamiseen, jota on havainnollistettu kuvassa 6. Olio-ohjelmoinnin tapa ei olekaan muistin kannalta kovin tehokas tapa käsitellä tietoa, sillä samankaltaiset tiedot ovat sirpaloituneena eri osiin muistia. (Llopis 2009).




Kuva 6. Olioihin ryhmiteltyjen tietorakenteiden kutsujärjestys (Llopis 2009)

Paras tietorakenteiden sijoittelu saadaan hajottamalla tietorakenteet pienikokoisiin komponentteihin. Nämä komponentit on helpompi ryhmitellä yhteen ja luoda muistiin palasia, joihin on tallennettu saman tyyppistä tietoa. Tämä vähentää tiedon ylimääräistä muuttamista ja mahdollistaa nopean samanlaisen tiedon prosessoinnin perätysten. Tällä tavalla järjesteltyä tietoa havainnollistetaan kuvassa 7. (Llopis 2009).




Kuva 7. Komponenttilistoihin ryhmiteltyjen tietorakenteiden kutsujärjestys (Llopis, 2009)

Olio-ohjelmointi perustuu koodiorientoituneisiin suunnittelumenetelmiin, joissa suunnittelun pohja on ongelmissa ja niiden ratkaisumenetelmissä. Vastaavasti dataorientoitunut suunnittelu keskittyy lopputulokseen. Tämä tarkoittaa sitä, että koodipainotteisen suunnittelun lähtökohta on suunnitella paras mahdollinen menetelmä tuottaa idea tai ominaisuus selkeästi. Täten edetään järjestyksessä syötteestä kohti lopputulostetta pyrkimyksenä mahdollisimman luettava ja tehokas koodi. Dataorientoituneessa suunnittelussa puolestaan mietitään käytettävien menetelmien sijaan, minkälainen lopputuloste halutaan tietynlaisella syötteellä. Tämän jälkeen koodia aletaan rakentaa niin sanotusti väärinpäin edeten lopusta kohti alkua ja alkuperäistä syötettä. Keskittymällä haluttuun lopputulokseen saadaan usein kehitettyä tapa muokata tietoa mahdollisimman vähillä tietorakenteiden muutoksilla. Näin säästetään muistin ylimääräistä rasitusta. (Llopis 2009; Collin 2010; Mitton 2015).

## 3.2 Hyödyt ja ongelmat

Dataorientoitunut suunnittelu loistaa siellä, missä olio-ohjelmointi nykyään epäonnistuu. DOD-malli vastaa moneen ongelmaan suunnittelunsa puolesta, joita nyky-

aikaiset teknologiset ratkaisut ovat luoneet. Yksi suurimmista eduista on, että moniajo on helpompaa. Jotta moniajo olisi mahdollista olio-ohjelmoinnissa, vaatii se paljon asioiden synkronoimista, mikä johtaa tehokkuuden katoamiseen. Olio-ohjelmoinnissa säikeet helposti jäävät odottamaan, että toiset säikeet saisivat prosessoitua tiedon, jota tarvittaisiin. Dataorientoituneessa suunnittelussa tämä ei ole niin suuri ongelma, sillä siinä funktiot ovat pieniä ja toisistaan riippumattomia. Funktiot on toteutettu niin, että ne käsittelevät pelkästään syötteen ja palauttavat sitten tulosteen kutsujalle. Funktion ulkopuolista tietoa ei käytetä ollenkaan mikä estää funktioiden välisten riippuvuuksien syntymistä. Tällaiset funktiot, jotka eivät ole toisistaan riippuvaisia, on helppo jakaa eri säikeille tehtäväksi, eikä niitä juurikaan tarvitse synkronoida toisiinsa nähden. (Llopis 2009; Fabian 2013)

Sen lisäksi, että funktiot ovat toisistaan riippumattomia ja pieniä kokonaisuuksia, rinnakkaisen ohjelmoinnin toteutusta helpottaa myös tiedon jakaminen komponentteihin. Niihin jäsennelty tieto on helpompi jakaa kokonaisuuksiksi joiden välillä ei ole riippuvaisuuksia ja joita yksi ydin voisi kerrallaan käsitellä. Poistamalla riippuvuudet eri tietorakennekokonaisuuksien väliltä minimoidaan myös riippuvuuksista syntyvät ongelmat, kuten kilpailutilanne ja lukkiutuminen. Jaettua tietoa eri säikeiden välillä joudutaan kuitenkin joissain tapauksissa käyttämään. Tällaisissa tapauksissa säieturvallisuuden voi taata niin, että funktioissa ei käytetä muita tietorakenteita, kuin mitä niille on parametreissa annettu. Jos funktio kuitenkin käsittelee ulkopuolisia muuttujia, on tärkeää muistaa lukita muuttuja, jotta muut säikeet eivät voisi sitä samaan aikaan käyttää ja vapauttaa se sen jälkeen, kun tarvittavat operaatiot muuttujalle on tehty. (Llopis 2009; Suleman 2011; Fabian 2013)

Toinen suuri dataorientoituneen suunnittelun tuoma hyöty on, että se on paljon tehokkaampi tapa lähestyä yhtä nykyteknologian suurinta kompastuskiveä, hidasta muistia. Olio-ohjelmoinnin pakatessa tietorakenteita ylisuurten luokkien sisään, jota muistit eivät osaa kunnolla prosessoida, dataorientoitunut suunnittelu järjestelee tiedon järkevästi. Tämä nopeuttaa tiedon prosessointia ja välimuisti tulee käyttöön tehokkaammin. Lisäksi toisistaan erotellut tietorakenteet ja logiikka tekevät systeemistä huomattavasti modulaarisemman, sillä erilaiset riippuvuudet eri asioiden välillä vähenevät. Lisäksi erillään olevia tietorakenteita ja logiikkaa on myöhemmin helppo muokata ja uutta koodia on helpompi luoda. (Llopis 2009)

Dataorientoituneen suunnittelun suurin haittapuoli on itse konseptin vaikeus. Sitä ei ole niin helppo opettaa kuin esimerkiksi olio-ohjelmointia, sillä DOD-malli ei ole millään tavalla mallinnettavissa reaalimaailmaan, eikä sille ole helppo kehittää muitakaan konsepteja, joita käyttää hyödyksi opetettaessa. Asiaa ei auta se, että suurimmalla osalla ohjelmoijista on vahva OOP-tausta ja kouluissa opetetaan ensin muita ajattelutapoja. Dataorientoitunut suunnittelu vaatiikin ajattelumallin täydellistä muuttamista. Muita haittoja on se, että DOD-malli ei toimi oikein yksi yhteen olio-ohjelmoinnin kanssa, vaan niiden suunnitteluperiaatteet törmäävät nopeasti. Näinpä dataorientoitunutta suunnittelua kannattaa käyttää kirjoittamalla koko systeemi sen periaatteiden mukaisesti. Puhtaan DOD-systeemin luominen on kuitenkin vastaavaa OOP-systeemiä huomattavasti haastavampi ja raskaampi luoda. Täten on järkevää arvioida, saadaanko DOD-pohjasta enemmän etuja, jotta koko arkkitehtuuri olisi järkevää luoda sen periaatteiden mukaisesti. Tähän valintaan vaikuttavat projektin koko, aikataulu ja budjetti. (Llopis 2009; Sefton 2016)

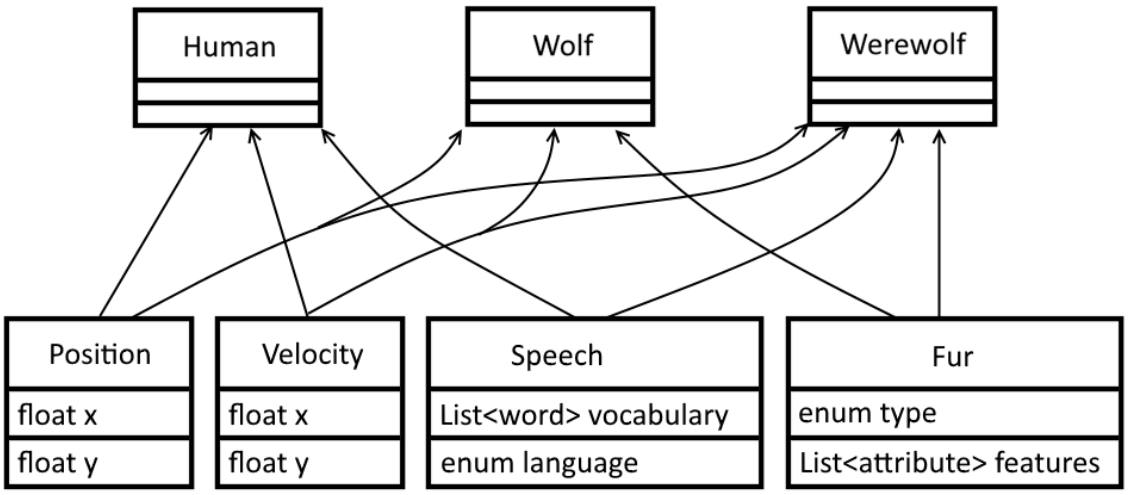
## 3.3 Komponenttimalli

Komponenttimalli on perinnän lisäksi toinen tapa toteuttaa oliot, tai entiteetit. Komponenttimalli on viime aikoina ollut suuressa suosiossa käytettävästä ohjelmointiparadigmasta riippumatta. Tämä on nähtävissä pelimoottoreista kertovissa artikkeleissa ja peliprojektien post-mortemeissa, joissa yleinen teema on ollut siirtyminen perintämallista komponenttimalliin. Esimerkiksi Unityn arkkitehtuuri perustuu vahvasti olio-ohjelmointiin ja komponenttimalliin. Komponenttimallissa olio-ohjelmoinnin olio on itsenäisen tietorakenneyksikön sijaan säiliö pienemmille objekteille, komponenteille. Säiliöistä voidaan lukea entiteetin kaikki arvot suoraan. (Porter 2012; Unity Technologies 2017)

*Thief: The Dark Project* oli yksi ensimmäisistä komponenttimallia hyödyntävistä peleistä. Projektissa kehittäjien tavoitteena oli luoda työkalut, jotka mahdollistivat työntekijöiden mahdollisimman itsenäisen työskentelyn. Tämä saavutettiin keskittymällä pelissä mahdollisimman paljon tiedon muokkauksen helpottamiseen. Näin myös muut kuin ohjelmoijat implementoimaan asioita, sillä he pystyivät muuttamaan peliä vain muokkaamalla tekstitiedostoja. Pelimoottori pystyi lukemaan näitä

tekstitiedostoja, jotka suorittivat projektissa komponenttien virkaa. Tämä mahdollisesti sen, että asioiden muokkaaminen ei vaatinut muutoksia koodiin. Yksi komponenttimallin hyödyllisyyksistä, varsinkin toimiessaan DOD:in rinnalla on, uuden sisällön tuottamisen ja vanhan muokkaamisen helppous, kun systeemi on rakennettu valmiiksi. Valmiit komponenttikokonaisuudet luovat myös jatkumoa, sillä niitä voi käyttää myöhemmissäkin projekteissa. (Leonard 1998; Porter 2012.)

Komponenttipohjaisessa toteutuksessa luokat ovat entiteettejä, joihin on säilötty komponentteja. Komponentti hoitaa yhden toiminnan, esimerkiksi liikkumisen. Entiteetti määrittää mitä komponentteja siinä on, joka vähentää konflikteja ja päällekkäistä tietoa. Jos eri entiteeteille halutaan samalla logiikalla tapahtuva toiminto, voidaan samaa komponenttia voi käyttää useassa eri entiteetissä. Esimerkiksi objektin sijainti ja nopeus voidaan jakaa erillisiin komponentteihin, jolloin ainoastaan liikkuvilla objekteilla on nopeuskomponentti. Aiemmin kappaleessa 2.3 esitetty ongelma Human-Wolf-Werewolf-perintätilanteessa voidaan selvittää yksinkertaisesti komponenteilla. Ongelma ratkeaa siten, että tehdään jokaisesta ominaisuudesta oma komponenttinsa, jolloin perimistä ei tarvitse käyttää eikä mikään entiteetti käytä ylimääräisiä komponentteja – pelkästään niitä mitä entiteetti tarvitsee. Entiteettien komponenttipohjausta toteutustapaa on havainnollistettu seuraavassa kuvassa.




Kuva 8. Komponenttipohjainen olioiden toteutus

Komponenttipohjainen olion toteutustapa ratkaisee moniperintää paremmin perinnässä esiintyvät ongelmat. Moniperinnän luodessa uusia ongelmia perinnässä

esiintyvien ongelmien päälle, kuten timanttiongelman, komponenttipohjaisessa toteutuksessa niitä ei juuri esiinny. Kun eri ominaisuudet on jaoteltu omiksi modulaarisiksi kokonaisuuksikseen, joita oliot voivat sisällyttää itseensä, on olioiden luominen helpompaa. Seuraavassa kuvassa on havainnollistettu, miten komponentin voi luoda C#:ssa hyödyksi käyttäen palveluliittymiä. (Boreal Games 2013)

```
public class PositionComponent : IComponent
{
     float x;
     float y;
     float z;
}
```

Kuva 9. Esimerkki komponentin toteutuksesta C#:ssa

Komponenttimallia varten ei tarvitse miettiä ja toteuttaa monimutkaista perintähierarkiaa, jotta mahdollisimman paljon koodia saataisiin käytettyä uusiksi. Sen sijaan voidaan luoda useita komponentteja, joita lisätään olioihin ja näin luodaan uusia olioita. Kun perintää ei ole käytetty, perinnässä ongelmana oleva muokkauksen jäykkyys vältetään, sillä uusiksi kirjoitettu komponentti rikkoo vain itsensä ja sitä käyttävät oliot, toisin kuin isäntäluokan muokkaus, joka rikkoo kaikki alla olevat lapsiluokat herkästi. Olioita luotaessa ei tarvitse miettiä minkä luokan lapsena uusi olio toimisi, vaan jokainen olio on oma kokonaisuutensa. Oliota luodessa mietitään vain mitä tietoa olion pitää sisältää, eli mitä komponentteja sen tulee käyttää. (Boreal Games 2013)

Kun tiedot jaetaan komponenteiksi vältetään se, että olioon on sisällytetty turhaa tietoa. Tämä on yleinen ongelma perinnässä, jossa lapsiluokka ei aina tarvitse kaikkea, mitä isäntäluokassa on toteutettu. Uusien ominaisuuksien lisääminen on helpompaa, sillä niitä varten voi luoda vain uuden komponentin, jonka oliot voivat ottaa käyttöönsä. Jos perintämallissa tarvitaan uusi ominaisuus, jota monet oliot tarvitsevat, on se järkevin lisätä oikeaan isäntäluokkaan. Oikean isäntäluokan valitseminen tosin voi olla vaikeaa, ja sen lisäksi pahimmassa tapauksessa, kuten edellä on mainittu, isäntäluokan muokkaus voi rikkoa siitä periytyviä luokkia. (Boreal Games 2013)

Vaikka komponenttimallia on mahdollista hyödyntää olio-ohjelmoinnissa olioiden toteutukseen, on komponenttimalli ajatusmalliltaan käyttökelpoisempi dataorientoituneessa suunnittelussa. Olio-ohjelmoinnissa komponentit sisältävät tietorakenteiden lisäksi myös niitä käsittelevän logiikan, eli jokainen komponentti muokkaa funktioilla sisällään olevaa tietoa. Dataorientoituneessa suunnittelussa vastaavasti komponentit sisältävät pelkästään tietorakenteita, ja niitä käsittelevät logiikat on toteutettu muualla. Tämä muuttaa komponenttimallin modulaarisemmaksi, sillä tietorakenteet ja logiikka eivät näin enää ole riippuvaisia toisistaan. Komponenttimalli ei ole suoraan dataorientoituneen suunnittelun kanssa käsi kädessä kulkeva käsite, vaan se on eräänlainen esiaskel kohti täydellistä DOD-mallin toteutusta. Dataorientoitunut suunnittelu täydentää komponenttimallin perusteita, että lopullinen toteutus saavutetaan. (Porter 2012; Fabian 2013)

## 3.4 Tietorakenteiden jakaminen komponentteihin

Perinteisessä olio-ohjelmoinnissa tietorakenteet on tallennettu joko kokonaan olioihin itseensä perintämallilla tai komponenttimallin mukaisesti. Komponenttimallisessa toteutuksessa tietorakenteet on hajautettu pieniin säiliöihin, komponentteihin, jotka ovat tämän jälkeen säilötty olioon. Dataorientoitunut suunnittelu toteuttaa jälkimmäistä mallia vieden sen ajatusta astetta pitemmälle. Dataorientoituneessa suunnittelussa käsite olio, myöhemmin entiteetti, on häivytetty lähes kokonaan. Säiliönä komponenteille toimimisen sijasta entiteetti on käytännössä vain ID-muuttuja, jonka avulla eri listoihin tallennetut komponentit ovat tunnistettavissa kuuluvaksi samaan kokonaisuuteen. Entiteettikäsitteen poistamisen lisäksi dataorientoituneessa suunnittelussa komponentit eivät sisällä niitä muokkaavia logiikkoja, vaan niihin on säilötty pelkästään tietoa. Logiikat on toteutettu muualla omina kokonaisuuksinaan. (Fabian 2013)

Jokaisessa komponenttimallin toteutuksessa komponenttien tallentaminen listaan on suotavaa. Tämä helpottaa komponenttien hallitsemista toteutustavasta riippumatta, sillä komponentteihin käsiksi pääsy on helpompaa ja erilaista tietoa voidaan käydä läpi helposti silmukassa. Olio-ohjelmointi tyyppisessä toteutuksessa komponentit voidaan listata olion sisällä. Komponenttien periytyessä itse tehdystä

IComponent-palveluliittymästä tallentaminen listaan on mahdollista. Jos myös logiikka on toteutettu komponentissa, voidaan tämän jälkeen helposti päivittää kaikki oliossa olevat komponentit niissä olevalla Update-funktiolla, joka periytetään IComponent-palveluliittymästä. Tätä komponenttilistan toteutustapaa on havainnoitu kuvassa 10 C#:lla. (Microsoft 2017; Fabian 2013)

```
public class GravityComponent : IComponent
{
    MovementComponent movement;
    public Vector3 gravity;
    bool isGrounded;

    void Update() {
        if (!isGrounded) {
            movement.velocity += gravity;
        }
    }
}

public class PlayerEntity
{
    MovementComponent movement;
    GravityComponent gravity;
    public Dictionary<Component, IComponent> components;

    void Update() {
        foreach (IComponent component in components) {
            component.Update();
        }
    }
}

enum Component
{
    Gravity,
    Movement,
    Count
}
```

Kuva 10. Logiikan sisältävien komponenttien toteutus ja listaus olioon

Kuten yllä olevasta toteutuksesta huomataan, logiikan ja tietorakenteet sisältävä komponentti ei ole kovin modulaarinen. Kun sekä logiikka että tietorakenteet ovat kiinni komponenteissa, joutuu osa komponenteista todennäköisesti hakemaan viittauksia toisiin komponentteihin ja muokkaamaan niissä olevaa tietoa. Tämä tekee komponentit riippuvaiseksi toisistaan, eivätkä komponentit voisi toimia itsekseen, ilman toisten komponenttien olemassaoloa. Viittauksien hakeminen on myös työlästä ja itse viittausmekaniikka voi olla koodin lukijalle epäselkeää. Koodi itsessään ei välttämättä kerro lukijalle minkä entiteetin komponenttia nykyinen komponentti käyttää toimiakseen tai mitä tietoa se ylipäätään muokkaa.

Logiikat, joita ei ole toteutettu komponenttien sisällä, voivat päästä käsiksi entiteettien komponentteihin hakemalla niitä entiteetin komponenttilistasta oikealla avaimella tai komponentin tyypillä. Näin logiikkojen ei tarvitse käyttää itse entiteettiä muuhun kuin komponentin viittauksen hakuun. Hyvin toteutetulla hakumenetelmällä logiikan ei tarvitse tietää mitä luokkaa entiteetit ovat. Tämä on mahdollista esimerkiksi silloin, kun kaikki entiteetit periytyvät samasta entiteetti-isäntäluokasta. Isäntäluokassa voidaan tallentaa IComponent-palveluliittymätyyppiä oleva lista ja siihen voidaan toteuttaa komponentin hakualgoritmi, jotka molemmat automaattisesti periytyvät lapsientiteeteille. Tätä entiteetin komponenttilistan toteutustapaa ja komponentin hakumenetelmää esitellään kuvassa 11 koodikielenä C#. (Fabian 2013)

```
public class GravityComponent : IComponent
{
     public bool isGrounded;
     public Vector3 gravity;
}

public class BaseEntity
{
     public Dictionary<Component, IComponent> components;
     public IComponent GetComponent(Component type){
          IComponent component;
          if(!components.TryGetValue(type, out component)) {
               throw new Exception(
                    "Didn't find component of type", type);
                    return null;
          }
          return component;
     }
}

public class GravityLogic
{
     MovementComponent movementComponent;
     GravityComponent gravityComponent;

     public void GetReferences(){
          BaseEntity playerEntity;
          movementComponent = MovementComponent)playerEntity.
          GetComponent(Component.Movement);
          gravityComponent = (GravityComponent)
          playerEntity.GetComponent(Component.Gravity);
     }

     void Update(){
          movementComponent.velocity += gravityComponent.gravity;
     }
}
```

Kuva 11. Komponenttien toteutus ja listaus entiteettiin, jotka sisältävät vain tietoa

Puhtaasti dataorientoituneessa toteutustavassa komponenttien ei tarvitse periytyä IComponent-palveluliittymästä vaan jokaista komponenttityyppiä varten on luotu oma listansa, joihin komponentit tallennetaan. Logiikat tuntevat komponenttilistat ja käsittelevät niitä joko kaikki kerralla silmukassa, tai tiettyä haluttua komponenttia entiteetin ID-tunnisteen avulla. Komponenttien lisäksi myös yksittäisiä muuttujatyyppejä voi ja usein myös on järkevää listata. Yhden muuttujatyypin listojen käyttäminen pakkaa samanlaista tietoa samoihin muistin osiin erittäin tehokkaasti. Kuvassa 12 on havainnoitu puhtaasti dataorientoitunutta mallia toteuttavassa komponenttilistassa C#:ssa. (Fabian 2013)

```
public class GravityComponent
{
     public bool isGrounded;
     public Vector3 gravity;
}

public static class ComponentManager
{
     public Dictionary<int, GravityComponent> gravityComponents;
     public Dictionary<int, MovementComponent> movementComponents;
     public Dictionary<int, DamageComponent> damageComponents;
     .
     .
     .
}

public class GravityManager
{
     Dictionary<int, GravityComponent> gravityComponents =
          ComponentManager.gravityComponents;
     Dictionary<int, MovementComponent> movementComponents =
          ComponentManager.movementComponents;

     void Update() {
          foreach (KeyValuePair<int, GravityComponent>
               pair in gravityComponents) {
               GravityComponent gravityComponent = pair.Value;
               if (!gravityComponent.isGrounded) {
                    MovementComponent movementComponent;
                    if (movementComponents.
                         TryGetValue(pair.Key, out movementComponent)) {
                         movementComponent.velocity +=
                              gravityComponent.gravity;
                    }
               }
          }
     }
}
```

Kuva 12. Komponenttien toteutus ja listaus dataorientoituneessa suunnittelussa

Dataorientoituneessa suunnittelussa komponentit on tallennettu listoihin, joita logiikat voivat helposti käydä läpi silmukassa. Lista on tyypitetty komponenttiluokalla

ja yleensä se on dynaaminen, jolloin komponentin listaan lisääminen on joustavaa. Eri tyyppisten komponenttien ollessa erillään omissa listoissaan, on ne mahdollista sitoa toisiinsa omistajaentiteetin ID-tunnisteella, joka annetaan komponenteille automaattisesti niitä luotaessa. Tunnisteen avulla haluttu komponentti on myös mahdollista etsiä komponenttilistasta. Komponenttien ollessa selkeästi erotellut, on niiden käyttäminen erittäin modulaarista ja nopeaa. Esimerkiksi haettaessa entiteetin sijaintia sijainnit-listasta saadaan vain entiteetin sijainti eikä yhtään ylimääräistä tietoa. Komponenttien lisäksi on myös yksittäisistä muuttujatyypeistä, kuten boolean-tyypistä, mahdollista tehdä omia listojaan ja käyttää niitä kuin komponenttilistoja. (Fabian 2013)

Uuden entiteetin luominen on nopeaa dataorientoituneessa suunnittelussa, sillä se tarvitsee alustaa vain ID-tunnisteen kanssa. Tämän jälkeen lisätään sille tarpeelliset komponentit listoihin, jotka yhdistyvät automaattisesti entiteettiin tunnisteen avulla. Erilaisten entiteettien luominen on helppoa ja joustavaa entiteetin itsessään ollessa vain tunnistearvo. Uuden entiteetin luomista on havainnollistettu kuvassa 13. Entiteetti voi sisältää ihan mitä tahansa komponentteja ilman rajoituksia, mikä lisää joustavuutta entiteettien luomiselle. Erillisten komponenttien lisääminen ja poistaminen entiteetteihin on sujuvaa kaikkien komponenttien ollessa tietorakenteina toisistaan riippumattomia. Toisen komponentin muokkaus muuttaa toista komponenttia, vain välikätenä toimivan logiikan kautta, eikä muokkaus koskaan voi rikkoa toista komponenttia. (Fabian 2013; Llopis 2010)

```
public int CreateNewPlayerEntity() {
    int entityID = GenerateID();
    gravityComponents.Add(entityID, new GravityComponent());
    movementComponent.Add(entityID, new MovementComponent());
    positions.Add(entityID, Vector3.zero);

    return entityID;
}
```

Kuva 13. Uuden entiteetin luonti dataorientoituneessa suunnittelussa

Dataorientoineella suunnittelulla saadaan tietorakenteet toteutettua yksinkertaisina ja modulaarisina paketteina. Toiminnot ovat pieniä ja niissä on vähän tai ei ollenkaan sidonnaisuuksia muihin komponentteihin. Pienet komponentit ovat

myös helposti ymmärrettäviä ja päivitettäviä. DOD:lla toteutetussa arkkitehtuurissa muistipalaset koostuvat saman tyyppisistä peräkkäisistä tietorakenteista, joka parantaa hitaan välimuistin käyttöastetta ja nostaa ohjelman suorituskykyä merkittävästi. (Llopis 2009)

Pieniä tietorakennekokonaisuuksia on helppo siirtää ja muuttaa suoritettavaksi eri säikeille dataorientoituneessa suunnittelussa. Näin on helpompi jaotella tietyille prosessoriytimille tietynlaiset laskentatehtävät, mikä nostaa suorituskyvyn lisäksi myös ymmärrettävyyttä. Lisäksi kun jaetaan vain tietyntyyppisistä tietoa komponentille, esimerkiksi grafiikkaan liittyviä materiaaleja, niin tässä tapauksessa koneen on helpompi siirtää tieto pelkän näytönohjaimen prosessoitavaksi. Samalla prosessori voi prosessoida grafiikkaan linkitettyjä sijaintitietoja ja näin eri tietokoneen komponentit voivat toimia synkronoidusti ja saumattomasti yhdessä. (Llopis 2009)

Kun logiikkaa ei ole komponenteissa ollenkaan, tietorakenteet ovat kaikesta riippumattomia. Komponenttien käyttäminen on helpompaa, sillä niistä ei tarvitse kuin lukea ja tallentaa arvoja. Erillisten komponentin sisällä sijaitsevien funktioiden kutsulle ei ole tarvetta. Tehokkuusetujen lisäksi tämä siistii kirjoitettua koodia, sillä ei ole tarvetta kirjoittaa pitkiä funktiokutsuja, esimerkiksi entiteetissä olevasta komponentista kutsuttava päivitä-funktio: Entity.component[index].Update(). Logiikkojen, jotka ovat komponenteissa syvällä, välinen keskustelu on työlästä toteuttaa. (Llopis 2009; Fabian 2013)

## 3.5 Logiikan erottaminen omaksi kokonaisuudekseen

Dataorientoituneessa suunnittelussa logiikka on erotettu tietorakenteista. DOD-mallissa logiikat on muotoiltu eräänlaisiksi hallintaluokiksi, joilla on viittaus kaikista komponenttilistoista, joita niiden tarvitsee muokata. Dataorientoituneen suunnittelun peruspiirteisiin kuuluu, että yksi hallintaluokka suorittaa vain yhtä tehtävää, kuten myös yhdessä komponentissa on vain siihen liittyvää tietoa, ei mitään ylimääräistä. Pieniä hallintaluokkia on helpompi lisätä ja muokata, kuin monimutkaisia.

Pieniin osiin jaettujen logiikkapalasten modulaarinen etu on siis selkeä. (Fabian 2013; Llopis 2009)

Hallintaluokka muokkaa vain tietoja, joihin sillä on viittaukset. Yksi tapa antaa hallintaluokan funktiolle viittaukset, on luovuttaa ne parametreina. Kun ohjelma kutsuu jollain tavalla hallintaluokassa olevaa funktiota, se antaa parametreina informaation mitä tietorakenteita kutsuja toivoo funktion muokkaavan, sekä informaation, jota käytetään tietorakenteiden muokkaamiseen. Informaatio muokattavasta tiedosta voi olla käyttötarpeesta, käyttöpaikasta ja ohjelmointityylistä riippuen pelkkä entiteetin tunnistenumero, entiteettiryhmä, yksittäinen komponentti tai komponenttilista. Esimerkit tällaisista funktioista on esitelty kuvissa 14 ja 15.

```
public void AddForce(int[] entityIndexes, Vector3[] directions,
     float initialSpeed, float[] distancesFromAoe,
     float speedMultiplier, int count) {
          for(int i = 0; i < count; i++){
               movementComponents[entityIndexes[i]].velocity +=
               directions[i] * initialSpeed + (distancesFromAoe[i] *
               speedMultiplier);
     }
}
```

Kuva 14. Funktio, joka muokkaa määrätyn entiteettijoukkion nopeutta

```
public void AddForce(MovementComponent[] movementComponets, Vector3[]
     directions, float initialSpeed, float[] distancesFromAoe, float
     speedMultiplier, int count) {
     for (int i = 0; i < count; i++) {
          movementComponents[i].velocity += directions[i] * initialSpeed
          + (distancesFromAoe[i] * speedMultiplier);
    }
}
```

Kuva 15. Funktio, joka muokkaa määrätyn komponenttijoukkion nopeutta

Monien hallintaluokkien pitää muuttaa tietoa jokaisella kehyksellä. Täten hallintaluokkiin on asetettu viittaukset komponenttilistoista, joiden tietoa ne tulevat muokkaamaan säännöllisesti. Itse komponenttilistat on yleensä tallennettu julkiseen komponenttilistojenhallintaluokkaan, joten tarvetta erillisille etsimisalgoritmeille ei ole, jotta logiikkahallintaluokat pääsisivät oikeisiin listoihin käsiksi. Joka kehys ajettavaa hallintaluokan funktiota, komponenttilistojen tallennuspaikkaa, sekä komponenttilistojen viittauksen tallentamista hallintaluokkaan on havainnollistettu kappaleen 3.4 kuvassa 12. Oli tarvittavat viittaukset haettu tavalla tai toisella, logiikka

suorittaa silmukan, jossa se käy läpi komponenttilistan, jonka tietoa halutaan ensisijaisesti muuttaa. Jos muita komponentteja ei vaadita, hallintaluokka muuttaa kaikkien komponenttilistassa olevien komponenttien tietorakenteita saadun informaation avulla. Jos muita komponenttityyppejä tarvitaan tietorakenteiden muokkaukseen, logiikka muuttaa vain niitä listan komponentteja, jotka on linkitetty muihin tarvittaviin komponenttityyppeihin entiteetin tunnisteen avulla. (DICE 2010; Fabian 2013)

Hallintaluokka pystyy muokkaamaan komponenttilistan komponentissa olevaa tietoa vain, jos se löytää kaikki tarvitsemansa komponentit muista listoista linkitettynä samalla entiteettitunnisteella. Esimerkiksi piirtämistä hallitseva luokka piirtäisi ruutuun entiteetin vain, jos sen tunnisteella löytyy komponentti sekä sijainti-, että piirrettävät objektit-luokasta. Hallintaluokka voi myös käsitellä vain yhdenlaista tietoa, jolloin se ei tarvitsee toimiakseen useita saman tunnisteen omaavia komponentteja eri komponenttilistoista. Dataorientoituneen suunnittelun etu on se, että logiikan ja tiedon riippuvuus toisiinsa nähden on vain yksipuolinen. Tietorakenteet ovat olemassa ja käytettävissä, vaikka ohjelmaa varten ei olisi kirjoitettu yhtään luokkaa muokkaamaan tai käyttämään niitä. (Fabian 2013; Llopis 2010)

Hyvä dataorientoituneen suunnittelun mukainen hallintaluokka ei tarvitse toimiakseen toisia hallintaluokkia, tai tietoa niistä. Toimiakseen hallintaluokka tarvitsee vain viittaukset halutuista komponenttilistoista. Tällainen systeemi on hyvin modulaarinen. Keskustellakseen toisten hallintaluokkien kanssa arkkitehtuuriin on hyvä toteuttaa tapahtumahallintaluokka. Muut hallintaluokat rekisteröidään vastaanottamaan niitä tapahtumia, joista niiden tarvitsee tietää. Kun jokin toinen hallintaluokka haluaa lähettää informaatiota eteenpäin, se lähettää tapahtuman, jonka mukana lähetetään tietoa. Tämän jälkeen tapahtumahallintaluokka kutsuu tapahtumaa kaikilla muilla hallintaluokilla, jotka ovat rekisteröityneet tapahtuman kuuntelijoiksi. Hallintaluokat käyttävät tapahtuman tietoa ja esimerkiksi ajavat läpi funktion joka muokkaa tapahtuman pyytämiä tietorakenteita. Esimerkki tapahtumakuuntelijasta muuttamassa tietoa tapahtuman käynnistyessä on havainnollistettu kuvassa 16. (Fabian 2013)

```
public class GravityManager
{
    Action<int> eventListener;

    void GetReferences() {
        eventListener = new Action<int>(HitGround);
        EventHandler.StartListening(Event.HitGround, eventListener);
    }

    void HitGround(int entityIndex) {
        GravityComponent gravityComponent;
        if (gravityComponents.TryGetValue(entityIndex,
        out gravityComponent)) {
            gravityComponent.isGrounded = true;
        }
    }
}
```

Kuva 16. Tapahtumakuuntelija

Kun logiikka erotetaan tietorakenteista, ohjelman testaus helpottuu, koska logiikka on itsenäisesti toimiva kokonaisuutensa eikä siihen vaikuta muut objektissa olevat logiikat. Erillisille logiikoille on helpompi kirjoittaa automatisoituja yksikkötestejä, sillä logiikan toimintaa voi testata luomalla testisyötteen, antaa logiikan muokata sitä ja tarkistaa, että tuloste on halutunlaista. Etuna on myös se, että OOP-lähtökohtaan verrattuna DOD-paradigmalla tulee vähemmän välimuistiohituksia, koska tietokoneen välimuisti käsittelee tehokkaasti lineaarisia tietorakenteita. Dataorientoituneessa suunnittelussa yhdistyvät sekä koodin tehokkuus, että sen luettavuus ja ohjelman kehityksen helppous. Kun logiikka on kirjoitettu eritoten tietorakennelistojen muokkaamiseen, lopputuloksena on pieniä, helposti luettavia ja tehokkaita funktioita, jotka eivät ole riippuvaisia toisiinsa nähden. Tämä modulaarisuus on erittäin hyödyllistä nykyajan moniytimisille prosessoreille, sillä pienet logiikat on helppo jakaa eri ytimille ajettavaksi, varsinkin kun logiikkojen välillä ei ole riippuvaisuuksia. (Llopis 2009)

# 4 FUNKTIONAALINEN OHJELMOINTI

Funktionaalinen ohjelmointi perustuu vahvasti erilaisten laskelmien toteutuksiin, joihin se käyttää työkaluinaan funktioita. Funktioita käytetään lähestulkoon kaikkien toiminnollisuuksien toteuttamiseen - pienimmät laskut mukaan lukien. Tämä johtaa siihen, että käytännössä lähes kaikki arvot on funktionaalisessa ohjelmoinnissa korvattu funktioilla. Tämä onkin suurin ero funktionaalinen ja imperatiivisen paradigman välillä. Funktionaalinen ohjelmointi ei imperatiivisen tavoin tunne tilakäsitettä, eli muuttujat eivät varsinaisesti muuttujia, joiden arvoa muutetaan ohjelman edetessä ja näin viedään ohjelmaa eteenpäin. Sen sijaan funktionaalisessa ohjelmoinnissa luotu muuttuja on muuttumaton eikä sitä siis voi koskaan muuttaa jälkikäteen. Ohjelmaa viedään eteenpäin luomalla uusia muuttujia vanhojen perusteella. Funktiot ovat näiden arvojen luontiin käytettävä työkalu. (Atehwa 2013; Akhmechet 2006)

Puhtaat funktiot ovat yksi osa funktionaalista ohjelmointimallia. Nämä funktiot eivät koskaan ole riippuvaisia tiedosta, joka on tallennettu niiden ulkopuolelle. Sen lisäksi funktiot eivät vaikuta ulkopuolisiin tietorakenteisiin. Puhtaat funktiot ottavat tarvittavat tiedot parametreissa, tekevät siihen tarvittavat laskut ja sen jälkeen palauttavat uuden tiedon. (Akhmechet 2006; Atehwa 2013)

Funktionaalisessa ohjelmoinnissa pyritään jakamaan funktioita pieniksi apufunktioiksi, joita voidaan käyttää sisäkkäin ja peräkkäin toisissa funktioissa. Tällaista toimintaa kutsutaan funktioiden ketjuttamiseksi. Koska funktionaalisessa ohjelmoinnissa ei koskaan haluta muokata vanhaa tietoja, vaan luoda uutta tietoa, sisäkkäiset funktiot ovat hyödyllisiä uuden tiedon luomisessa. Esimerkiksi funktionaalisessa ohjelmoinnissa listoihin tallennettua tietoa ei muuteta iteroimalla ne läpi ja laskemalla muutokset iteroinnin aikana. Sen sijaan käytetään funktioita, jotka käyvät listan läpi ja päättävät, miten uusi tieto järjestyy. Nämä funktiot ottavat parametreinaan listan lisäksi myös sisäkkäisfunktion, joka tekee itse tiedon muuttamisen. Funktioita, jotka saavat parametrinaan funktion, kutsutaan korkeamman asteen funktioiksi eli funktionaaleiksi. (Akhmechet 2006; Atehwa 2013)

Kuten edellisessä kappaleessa on mainittu, funktionaalisessa ohjelmoinnissa listoihin tallennettua tietoa ei muuteta iteroimalla niitä läpi toistorakenteiden avulla. Tähän syynä on se, että toistorakenteessa tehtävä editointi muuttaisi tietorakennetta, mitä funktionaalisessa ohjelmoinnissa pyritään välttämään. Lisäksi toistorakeenteen päättyminen on riippuvainen eri muuttujien tiloista, kuten for-silmukka on riippuvainen kasvatettavan kokonaislukumuuttujan suuruudesta. Toistorakenteiden sijaan listan läpi käymiseen käytetään rekursiota. Rekursiofunktio on sellainen funktio, joka kutsuu itseään niin kauan, kunnes sen ehdot täyttyvät ja se voi palauttaa funktion tuloksen alkuperäiselle kutsujalle. Kutsuessaan itseään uudestaan, kasvattaa funktio aloittajaluku-parametrinsa kokoa, mikä vie listan läpi käymistä eteenpäin. Aloittajaluku-parametri suorittaa rekursiossa samaa virkaa, kuin toistorakenteiden kasvatettava kokonaisluku. (Atehwa 2013; Cook 2013)

## 4.1 Hyödyt ja ongelmat

Koska funktionaalinen ohjelmointi perustuu laskutoimitusten suoritukseen, ovat sen vahvuudet selkeästi kaikenlaisessa matemaattisessa ohjelmoinnissa kuten pelifysiikkojen laskennassa. Funktionaalinen ohjelmointi sopii kuitenkin hyvin kaikenlaisen tiedon prosessointiin ja muokkaamiseen. Tämä vahvuus johtuu erityisesti siitä, koska funktionaalisessa ohjelmoinnissa muuttujia ei voi muokata, vaan on aina luotava uusia. Täten vähemmän virhealtista koodia on helpompi kirjoittaa; on helpompi olla varma, että haluttu tieto on oikeassa muodossa. (Atehwa 2013)

Funktionaalisesti kirjoitetun koodin yksi selkeimmistä hyödyistä on, että sitä on helpompi yksikkötestata kuin imperatiivista koodia. Kun jokainen muuttuja on muuttumaton ja funktiot ovat riippumattomia ulkopuolisesta tiedosta, funktiota on helppo testata. Testissä ei tarvitse huolehtia funktioiden kutsujärjestyksestä, sillä funktiot ovat toisistaan riippumaattomia. Sitä varten ei myöskään tarvitse tehdä omia testaustiloja. Testatessa ainoa huolehdittava asia on, että funktion argumentit ovat oikeanlaiset. Funktion toimintaa onkin helppo testata antamalla sille rajatapausargumentit. (Akhmechet 2006)

Funktionaalisesti kirjoitetusta koodista on myös helpompi etsiä ja korjata virheitä. Myös tämä etu pohjautuu funktioiden riippumattomuuteen muusta koodista tai tietorakenteista. Jos funktio palauttaa väärän arvon, on arvo aina väärä riippumatta siitä mitä siihen parametrina syöttää. Näin vika on helpompi paikantaa, varsinkin jos kirjoitetut funktiot ovat varsin lyhyitä. Kun funktio on täysin toimiva, on sen lopputulos aina halutun lainen parametreista riippumatta. Näinpä funktionaalisista ohjelmista on helpompaa todistaa erilaisia ominaisuuksia, sillä itse koodia ei tarvitse muuttaa. (Akhmechet 2006; Atehwa 2013)

Funktionaalinen ohjelmointi ja siihen perustuvat kielet ovat yksi parhaista ratkaisuista nykyteknologialla syntyneeseen ongelmaan, moniajoon. Funktionaalisen ohjelmoinnin perusta keskittyy kiintoarvoihin, jotka ovat omiaan moniajossa syntyvään kilpailutilanteen ratkaisemiseen: Koska muuttujia ei muokata, vaan aina luodaan uusia tietorakenteita, toisistaan erossa olevat säikeet eivät muokkaa samoja tietorakenteita, eivätkä myöskään ylikirjoita toistensa muokkauksia. Täten ei myöskään tarvitse kirjoittaa koodiin ylimääräisiä säikeen lukitsemisia ja pelätä että syntyisi lukkiutumisia, jotka pysäyttäisivät ohjelman toiminnan. Itsekseen toimivien funktioiden synkronointia ei myöskään tarvitse huolehtia funktionaalisessa ohjelmoinnissa. (Akhmechet 2006)

Teho- ja käytännönhyötyjen lisäksi funktionaalisesti ohjelmoitu koodi on usein siistimmän näköistä ja luettavampaa kuin vastaava olio-ohjelmoitu koodi. Funktionaalisesti kirjoitetun ohjelman koodi on myös yleensä jonkin verran lyhempi kuin vastaavassa olio-ohjelmoidussa ohjelmassa olisi ollut. Näiden monien funktionaalisten ohjelmoinnin hyötyjen kannattajaksi on noussut muun muassa John Carmack, jonka mukaan (2012) funktionaalinen ohjelmointityyli antaa etuja aina, riippumatta siitä, että millä kielellä ohjelmoidaan. Funktionaalista tyyliä kannattaa käyttää hänen mukaansa aina kun se on käytännöllistä ja miettiä sen käyttöä tarkasti, kun se ei ole käytännöllistä. (Atehwa 2013; Carmack 2012.)

Funktionaalinen ohjelmointi ei ole monista selkeistä hyödyistään huolimatta kuitenkaan ratkaisu joka ongelmaan, myöskään peliohjelmoinnin saralla. Yksi suurimmista haittapuolista on, että funktionaalisessa ohjelmoinnissa voidaan menettää suorituskyvyn tehokkuutta. Funktionaalinen ohjelmointi laskee myös suoritus-

kyvyn ennakoitavuutta. Näihin vikoihin on kaksi syytä, jotka ovat suurempi roskienkeruun tarve ja muuttujien niin kutsuttu laiskuus. Nämä ongelmat ovat varsinkin pelien teossa kriittisiä, sillä pelit vaativat lähes tulkoon aina tasaisen ja hyvän suorituskyvyn. (Jelvis 2014)

Jatkuva uusien muuttujien luonti vanhojen muokkaamisen sijaan tekee roskienkeruusta paljon tarpeellisempaa ja enemmän aikaa vievää. Suuri osa uusista luoduista muuttujista on varsin lyhytikäisiä mikä kuormittaa roskienkerääjää jatkuvasti. Funktionaalisesti ohjelmoitua systeemiä varten on käytettävä paljon enemmän aikaa ja resursseja, jotta saataisiin luotua roskienkerääjä, joka ei tiputa ohjelman suorituskykyä merkittävästi, eikä epäsäännöllisesti. Roskienkerääjän tekemistä funktionaalisen kieleen vaikeuttaa myös se, että niissä ei ole olemassa heikkoa tarkistetaulua. Imperatiivisissa kielissä eri tyyppejä voidaan poistattaa samalla roskienkerääjällä heikon tarkistetaulun avulla, mutta funktionaalisissa kielissä liian erityyppisten muuttujien poistaminen pitää hoitaa toisista erillään. Tämä vaatii paljon monimutkaisemman roskienkerääjän. (Jelvis 2014; Carmack 2012; Harrop 2016)

Muuttujien laiskuus on ongelma suorituskyvylle, sillä se laskee sen ennakoituvuutta. Funktionaalisessa ohjelmoinnissa käytettävä laiska evaluaatio on hyödyllistä, sillä se laskee muistin varausta. Muuttuja ja täten muistia varataan aina vasta kun siinä olevaa informaatiota tarvitaan. Tämä siirtää myös funktion ajon kohtaan, jossa informaatiota tarvitaan. Tämä vaikeuttaa ajankohdan arvioimista, milloin todellinen laskenta tehdään. Lisäksi huonosti toteutettuna laiskasti evaluoidut raskaat funktiot voivat kerääntyä ajettavaksi samaan aikaan ja aiheuttaa satunnaista ruudunpäivityksen tippumista. Tämän estäminen on erittäin tärkeää peleissä. On myös vaikeampi analysoida, onko jossain tietyssä kohtaa ajoa haluttu funktio vielä suoritettu, tai onko se passiivisesti kuluttamassa muistia turhaan. (Jelvis 2014; Slava2006)

Funktionaalisten tietorakenteiden toteuttamisessa on lisäksi kaksi pienempää ongelmaa. Tehokkaiden tietorakenteiden suunnittelun ja toteutuksen näkökulmasta on merkittävä haitta, että funktionaalinen ohjelmointimalli ole suunniteltu siihen, että käytetään niin sanotusti tuhoavia päivityksiä, eli muutetaan muuttujien arvoja.

Tuhoavat päivitykset voivat olla vaarallisia väärinkäytettyinä, mutta erittäin hyödyllisiä, kun niitä käytetään oikein. Imperatiiviset tietorakenteet luottavat usein elintärkeillä tavoilla arvojen asettamiseen. Funktionaalisten tietorakenteiden toteutukseen täytyy löytää vaihtoehtoinen tapa. (Okasaki 1996, 1)

Ongelma funktionaalissa tietorakenteissa on myös se, että funktionaalisten tietorakenteiden oletetaan olevan joustavampia kuin imperatiivisten. Erityisesti kun imperatiivisia tietorakenteita päivitetään, vanhaan tietorakenteeseen ei tyypillisesti enää ole pääsyä. Kun funktionaalisia tietorakenteita päivitetään, oletetaan, että pääsy vanhaan ja uuteen tietorakenteeseen säilyy. Uudet tietorakenteet luodaan uusiin muuttujiin, joten vanhoihin muuttujiin tallennettu tieto pitäisi säilyä. Tietorakennetta, joka tukee useita versioita, kutsutaan pysyväksi ja vain yhtä versiota tukevaa tietorakennetta lyhytaikaiseksi. Funktionaalisissa ohjelmointikielissä tietorakenteet ovat tyypillisesti pysyviä, kun taas imperatiiviset tietorakenteet ovat tyypillisesti lyhytaikaisia. Funktionallisesti toteutetut pysyvät tietorakenteet voivat olla jopa tehottomampia kuin lyhytaikaiset tietorakenteet. Lisäksi, tutkimusten perusteella funktionaaliset ohjelmointikielet ovat pohjimmiltaan tehottomampia kuin imperatiiviset kielet joissain tapauksissa. Chris Okasakin mukaan on kuitenkin mahdollista toteuttaa funktionaaliset tietorakenteet niin, että ne ovat yhtä tehokkaita kuin parhaimmatkin imperatiiviset ratkaisut. (Okasaki 1996, 2)

Kuten dataorientoituneessa suunnittelussa, myös funktionaalisen ohjelmoinnin yksi suurista varjopuolista on, että itse konsepti on vaikea oppia ja se on hyvin erilainen verrattuna olio-ohjelmointiin. Varsinkin pelkän OOP-taustan omaavalle ohjelmoijalle funktionaalinen ohjelmointi on täynnä uusia suuria käsitteitä, kuten rekursio, muuttujien ainainen muuttumattomuus, laiskat tietorakenteet ja funktionaalinen reaktiivinen ohjelmointi. Funktionaalinen ohjelmointi vaatiikin hyvin erilaista ajattelu- ja lähestymistapaa kuin olio-ohjelmointi. (Jelvis 2014)

## 4.2 Funktiotyypit

Luonnollisesti, funktiot ovat funktionaalisen ohjelmoinnin pääkonsepti. Yksi funktionaalisen ohjelmoinnin eduista on, että siinä on mahdollista luoda ja käsitellä

funktiota kuten imperatiivisessa ohjelmoinnissa olioita. Funktionaalisessa ohjelmoinnissa funktiot ovat siis manipuloitavissa ja niitä on mahdollista lähettää parametreina toisiin funktioihin. Tämän johdosta saadaan lisää vapautta, sillä funktiot eivät ole osa mitään luokkaa, vaan itsenäisiä entiteettejä. C#:ssa funktio-entiteettejä voidaan käsitellä kuten muitakin tietorakenteita. (Popovic, 2012).

C#:ssa funktio-entiteeteillä täytyy olla tyyppi. Funktio-entiteetit ovat delegaatteja ja niitä on mahdollista tyypittää joko heikosti tai vahvasti C#:ssa. Delegaattien voidaan katsoa olevan funktion prototyypin määritelmä, missä kutsurajapinta on määritetty. Delegaattityyppien instanssimuuttujat ovat viittauksia funktioihin, kuten staattisiin metodeihin tai luokkametodeihin, joilla on prototyyppi. Kuvassa 17. on esimerkki tyypitetystä delegaattifunktiosta C#:ssa. (Popovic, 2012).

```
delegate double ExampleFunction(double x);
```

Kuva 17. Esimerkki delegaattifunktiosta

Edellisessä kuvassa esitetty delegaatti määrittelee prototyypin funktiosta, joka ottaa double-muuttujan parametrina ja palauttaa double-arvon. Huomioitavaa on, että vain tyypeillä on merkitystä toisin kuin metodien tai niiden argumenttien nimillä. Delegaattityyppi voisi olla trigonometrinen funktio, logaritmi-, eksponentti tai polynomilaskutoimitus tai mikä tahansa muu funktio. Kuvassa 18. esitellään esimerkki, jossa määritetään funktiomuuttuja, joka on viittaus matemaattiseen funktioon. Esimerkissä nähdään, miten funktion voi suorittaa kutsumalla funktiomuuttujaa ja näin tallentaa arvo johonkin tavalliseen muuttujaan. Esimerkissä nähdään myös, että funktiomuuttujan voi alustaa jälkikäteen uudella matemaattisella funktiolla. (Popovic, 2012)

```
ExampleFunction f = Math.Sin;
double y = f(4);
f = Math.Exp;
y = f(4);
```

Kuva 18. Esimerkki matemaattisen funktion asettamisesta tyypitettyyn funktiomuuttujaan

Vahvasti tyypitettyjen delegaattien sijaan on mahdollista käyttää myös geneerisiä funktiotyyppejä. Seuraavassa kuvassa 19. on uudelleen tehty edellinen merkki.

Siinä alustetaan funktiomuuttujan tyyppi heikosti vahvan funktiotyypityksen sijaan. Func-muuttujan tyypitykselle annetaan kaksi argumenttia, joista ensimmäinen on annettavan parametrin tyyppi ja toinen on palautettavan arvon tyyppi. (Popovic, 2012)

```
Func<double, double> f = Math.Sin;
double y = f(4);
f = Math.Exp;
y = f(4);
```

Kuva 19. Esimerkki matemaattisen funktion asettamisesta heikosti tyypitettyyn funktiomuuttujaan

Func-tyypin lisäksi C#:ssa funktio on mahdollista heikko tyypitys myös kahdella seuraavalla tyypillä:

- Predicate<T>
- Action<T1..Tn>

Predicate-tyyppi on heikosti tyypitetty funktiotyyppi, joka ottaa vastaan vain yhden argumentin. Annettu argumentti toimii annettavana parametrina, jonka perusteella funktio palauttaa kutsujalle boolean-arvon tosi tai epätosi. Seuraavassa kuvassa 20. on esitelty predikaattifunktio, joka ottaa vastaan string-muuttujan argumenttina. Funktiomuuttuja on alustettu funktiolla: String.IsNullOrEmpty. Tämä funktio ottaa vastaan string-arvon ja palauttaa tiedon siitä, onko annettu string-muuttujan arvo null tai onko se tyhjä. Täten se vastaa Predicate<string>-tyyppiä. Predicate<string>-tyypin sijaan voitaisiin käyttää myös Func<string, bool>-tyyppiä. (Popovic, 2012)

```
Predicate<string> isEmptyString = String.IsNullOrEmpty;
     if (isEmptyString("Test")) {
          throw new Exception("'Test' cannot be empty");
}
```

Kuva 20. Käyttöesimerkki heikosti tyypitetylle Predicate-funktiomuuttujalle

Action-tyypit ovat proseduureja, joita voidaan suorittaa. Ne hyväksyvät argumentteja, mutta eivät palauta arvoja. Seuraavassa kuvassa 21. hahmotetaan toiminto,

joka ottaa parametrina string-muuttujan. Tämä funktio alustetaan tavallisella Console.WriteLine-funktiolla. Kun toiminnon viittausta kutsutaan, kutsu välitetään Console.WriteLine-metodille. Kun käytetään Action-tyyppejä, määritetään vain lista parametreista, koska palautusarvoa ei ole. Esimerkki Action<string> vastaa Func<string, void>-tyyppiä. (Popovic, 2012)

```
Action<string> printLine = Console.WriteLine;
printLine("Test");
```

Kuva 21. Käyttöesimerkki heikosti tyypitetylle Action-funktiomuuttujalle

Listojen hallintaa funktionaalisessa ohjelmoinnissa hoitaa rekursio ja siihen pohjautuvat erilaiset funktiot. Tällaisia funktioita ovat esimerkiksi MapReduce-funktiot. Näissä funktioissa käsitellään listan sisältö läpi sisäkkäisfunktiolla, joka annetaan funktioon parametrina. Map-funktio ajaa jokaiselle listan osalle saman parametrina annetun sisäkkäisfunktion, luo uuden samankokoisen listan ja täyttää sen uusilla osilla. Uusi tieto riippuu alkuperäistiedosta ja siitä mitä sisäkkäisfunktio tekee niille. Map-funktion toimintaa on havainnollistettu kuvassa 22. Reduce-funktio vastaavasti yhdistää listoissa olevan informaation sisäkkäisfunktion toimiessa tiedon yhdistämisen määrittelijänä. Reduce-funktio palauttaa informaation summana uuteen arvoon. Esimerkki Reduce-funktiosta esitellään kuvassa 23. (Atehwa 2013; Cook 2013)

```
public IEnumerable<TResult> Map<T, TResult>(Func<T, TResult>
                    function, IEnumerable<T> list) {
     foreach (var i in list) {
          yield return function(i);
     }
}
```

Kuva 22. Map-funktio C#:ssa

```
int result = Reduce<int, int>(testFunction, originalList, 0);

public T Reduce<T, U>(Func<U, T, T> funtion, IEnumerable<U> list,
T accretion) {
     foreach (var i in list) {
          accretion = funtion(i, accretion);
     }
     return accretion;
}
```

Kuva 23. Reduce-funktio C#:ssa

## 4.3 Puhtaat funktiot

Puhdas funktio on funktio, joka käyttää vain sille syötettyä tietoa ja palauttaa arvon kutsujalle. Puhdas funktio ei muokkaa funktion ulkopuolista tietoa koskaan eikä myöskään sisällään luotuja muuttujia. Sen sijaan jokainen laskelma tallennetaan aina uuteen kiintoarvoon. Toteuttamalla funktiot puhtaasti saavutetaan kaksi selkeää etua: Niissä ei ole ollenkaan sivuvaikutuksia, koska funktio ei muuta muuttujia tai tietoa funktion ulkopuolella ja ne ovat johdonmukaisia, sillä ne antavat aina saman tulosteen, jos niille annetaan sama syöte. Koodin uudelleentoteuttaminen puhtaiden funktioiden avulla poistaa tarpeettomat sivuvaikutukset ja funktioiden väliset ulkoiset riippuvuudet. (Sturm, 2011. 29; Microsoft Corporation, 2015)

Epäpuhtaan funktion ero puhtaaseen funktioon on se, että epäpuhdas funktio muuttaa sisällään mitä tahansa arvoja. Kuvassa 24. olevalla koodilla havainnollistetaan epäpuhtaan funktion toimintaa. Koodi antaa tulosteeksi string-muuttujan: StringOne-StringTwo. ConcateString-funktio ei ole puhdas, koska se muuttaa sisällään funktion ulkopuolisen exampleText-muuttujan arvoa. (Microsoft Corporation, 2015).

```
public class Program {
     private static string exampleText = "StringOne";

     public static void ConcateString(string appendString) {
          exampleText += '-' + appendString;
     }

     public static void Main() {
          ConcateString("StringTwo");
          Console.WriteLine(exampleText);
     }
}
```

Kuva 24. Esimerkki epäpuhtaasta funktiosta

Puhdas funktio, kuten aiemmin on mainittu, on funktio, joka ei muuta sisällään mitään tietoa. Kuvassa 25. olevasta koodiesimerkistä huomataan selkeä ero puhtaan ja epäpuhtaan funktion välillä. Siinä ConcateString-funktio ei käytä muita muuttujia, kuin sille parametreina on annettu. Funktio ei myöskään muokkaa annettuja

parametreja mitenkään vaan palauttaa ne uuteen muuttujaan. Tuloksena on edellisen esimerkin kaltaisesti tuloste: StringOne-StringTwo. (Microsoft Corporation, 2015)

```
class Program {
     public static string ConcateString(string tempString,
               string appendString) {
          return (tempString + '-' + appendString);
     }

     public static void Main(string[] args) {
          string exampleTextA = "StringOne";
          string exampleTextB = ConcateString(exampleTextA,
                                   "StringTwo");
          Console.WriteLine(exampleTextB);
     }
}
```

Kuva 25. Esimerkki puhtaasta funktiosta

## 4.4 Tietorakenteet

Funktionaalisessa ohjelmoinnissa tietorakenteet ovat mielellään muuttumattomia. Kun ohjelmat kirjoitetaan käyttämään vain muuttumattomia tietorakenteita, metodit voivat ainoastaan palauttaa arvoja, eivät muokata niitä. Tämä johtaa usein siihen, että on helpompi nähdä tarkasti mitä ohjelman funktiot tekevät, sillä funktioiden ulkopuolisista muuttujista ei tarvitse huolehtia. Vaikka funktionaalisesti kirjoitetussa koodissa määritellyt tietorakenteet ovat muuttujia, antaa se hieman väärän käsityksen niistä, sillä tieto ei ole muuttuvaa. Tämän takia parempi termi muuttujalle, jonka arvoa ei vaihdeta sen jälkeen, kun se on kerran alustettu, on kiintoarvo. Muuttujien lisäksi funktionaaliset listat ovat muuttumattomia, joten listoissa olevia arvoja voidaan vain lukea. (Petricek 2009, 5-6)

C#:ssa voidaan määrittää muuttumattoman luokan tietoja avainsanalla *readonly*. Kiintoarvon voi alustaa C#:ssa käyttämällä määritettä const-määrite muuttujatyypin edellä. Tällöin kääntäjä ilmoittaa virheilmoituksella, mikäli muuttujaan, jolle on annettu const-määrite, yritetään alustuksen jälkeen tallentaa uutta arvoa. (Microsoft Corporation, 2015)

Funktionaalisen tietorakenteen toteuttamiseen C#:ssa on useita käyttökelpoisia muuttujatyyppejä, kuten esimerkiksi Tuple- ja Funclist-tyypit. Tuple on tietorakennetyyppi, jolle voidaan määrittää yhdestä kahdeksaan elementtiä. Funclist<T> on heikosti tyypitetty C#-luokka, joka voi säilöä minkä tyyppistä tietoa tahansa. C#:in listat saa toteutettua helposti muuttumattomina funktionaalisina listoina määrittämällä niiden asettajat yksityiseksi ja antamalla arvot konstruktorissa. Tämän vuoksi, kun lista on luotu, ei yhtään arvoa voida enää jälkikäteen muissa luokissa muuttaa. Hyväksikäyttämällä edellä mainittuja työkaluja voidaan C#:n luokassa saavuttaa täydellinen funktionaalinen toimivuus, jossa luokan ominaisuuksien muuttumattomuus on tärkeintä. Kuvassa 26 on esimerkki näin toteutetusta luokasta. (Petricek 2009, 60, 74, 180)

```
public sealed class Rect {
     public float Left { get; private set; } #A
     public float Top { get; private set; } #A
     public float Width { get; private set; } #A
     public float Height { get; private set; } #A
     public Rect(float left, float top, float width,
               float height) { #B
          Left = left; Top = top; Width = width; Height = height;
     }
     public Rect WithLeft(float left) { #1
          return new Rect(left, this.Top, this.Width,
                         this.Height); #C
     }
     // Similarly: WithTop, WithWidth and WithHeight #2
}
#A Readonly properties of the type
#B Construct the value
#1 Returns 'Rect' with modified 'Left' property
#C Create a copy of the object
#2 'With' methods for other properties (Omitted)
```

Kuva 26. Funktionaalisesti toteutettu C#-luokka

Edellisen esimerkin luokassa on käytetty yksityisiä set-komentoja ja julkisia get-komentoja. Koska luokan arvoja ei muuteta luokan luomisen jälkeen, ne ovat kiintoarvoja. Luokassa toteutetulla WithLeft-funktiolla voidaan luoda uusi kopio Left-ominaisuudesta ja tätä funktiota voidaan soveltaa myös muihin samankaltaisiin ominaisuuksiin. Hyöty tästä on se, että kaikkia Rect-luokan arvoja ei tarvitse lukea, vaan voidaan mainita pelkästään se ominaisuus, jota halutaan muokata. Seuraavassa kuvassa 27 on kuvattu, miten helposti arvoja voidaan tällaisilla funktioilla muokata. (Petricek 2009, 180-181)

```
var moved = rect.WithLeft(10.0f).WithTop(10.0f);
```

Kuva 27. Esimerkki pienistä funktioista, jotka on ketjutettu

String-objekti on kiintoarvo C#:ssa. Vaikka string-muuttujalle voidaan näennäisesti asettaa uusi arvo, on se todellisuudessa vain viittaus annetulle arvolle. Jokainen string-objekti, joka asetetaan string-muuttujaan, on tavallaan ilman new-avainsanaa toimiva olio, eli string-objektin kutsuminen palauttaa aina uuden objektin. Tästä syystä useat muutokset string-muuttujaan käsittelyyn pätevät samat heikkoudet kuin muihinkin funktionaalisen ohjelmoinnin tietorakenteisiin: useat muutokset ja tätä myötä uusien muuttujien luonti kuormittaa roskienkerääjää ja voivat aiheuttaa suorituskyvyn alenemista. (Microsoft, 2017)

Laiska evaluaatio on toimenpide, missä koodi suoritetaan vasta kun sitä tarvitaan ja ainoastaan niin monta kertaa kuin sitä tarvitaan - riippumatta siitä, missä kohti ohjelmaa koodi on. C#:ssa on muutamia viitekehyksiä, joilla ohjelman suoritusta voidaan tällä tavalla hallita, kuten IEnumerable ja IEnumerator. IEnumerable on pääluokka kaikille määritellyille kokoelmille, joita voidaan listata. Se sisältää yhden metodin, GetEnumeratorin, jolla voidaan palauttaa IEnumerator. Se mahdollistaa kokoelman iteroimisen Current-ominaisuudella sekä MoveNext- ja Reset-metodeilla. (Microsoft 2017)

Unityssä IENumerablea voidaan käyttää Coroutine-aliohjelman avulla. Aluksi määritetään IEnumerator-tyyppinen metodi, joka iteroi tiedot läpi. Uusi iteraatio palautetaan esimerkiksi komennoilla ”yield return null” tai ”yield return new WaitForSeconds(x)”, jossa x määrittää sekunneissa, miten kauan ohjelma odottaa, että uusi iteraation ajaminen aloitetaan. Coroutine-aliohjelman sisällä voidaan käyttää for-, foreach- tai while-silmukoita, kuten muissakin metodeissa. (Unity, 2017)

## 4.5 Rinnakkaislaskenta

Moniytimisiä prosessoreita hyödyntävä rinnakkaislaskenta tuo kirjoitettuun ohjelmaan paljon lisää suorituskykyä. Imperatiiviset kielet ovat pitkälti optimoituja yhteen ytimeen keskittyvään sarjaohjelmointiin, joten vaihtoehdot niille ovat olleet

tarpeen jo pari vuosikymmentä. Vaikka dataorientoitunut suunnittelu pystyykin pienellä vaivalla hyvin rinnakkaislaskentaan, on funktionaalisesti kirjoitettu koodi suvereeni tässä tehtävässä. Suurin etu funktionaalisten kielten rinnakkaislaskentakykyyn on se, että niissä muuttujat ovat useimmiten kiintoarvoja. Tämä vähentää kilpailutilanteiden riskiä huomattavasti. Koska kiintoarvoja ei voi edes sama säie muuttaa myöhemmin, eivät sitä ei voi muuttaa myöskään muut säikeet, mikä aiheuttaisi ristitilanteita. Kilpailutilanteen mahdollisuuksien eliminointi helpottaa funktionaalisesti ohjelmoimista, sillä koodiin ei koskaan tarvitse sekoittaa erillisiä muuttujien lukitsemisia, jotka pahimmillaan voivat aiheuttaa lukkiutumisen ajon aikana. (Milewski 2012, Akhmechet 2006)

Funktionaalisten kielten kääntäjät automaattisesti analysoivat funktiot, jotka voidaan antaa eri ytimien suoritettavaksi, vaikka ne olisi kirjoitettu sarjalaskennan mukaisesti. Jos jokin funktio tarvitsee informaatiota aiemmin suoritettavista funktioista, funktiota suorittava ydin odottaa automaattisesti, että kaikki tarpeellinen informaatio on prosessoitu, ennen kuin se aloittaa tehtävänsä. Kuvassa 28 on esimerkki, kuinka sarjamaisesti kirjoitettu koodi voitaisiin käytännössä jakaa kolmen ytimen suoritettavaksi. Imperatiivisissa kielissä tämä kääntäjän koodin optimisointi monelle ytimelle on mahdotonta, sillä funktiot eivät välttämättä ole puhtaita eli ne voivat muokata muuttujia, joita ei ole määritelty funktiossa itsessään. Funktioita seuraavat funktiot voivat myös olla riippuvaisia aiemmista, eikä imperatiivisten kielten kääntäjät osaa hallita automaattisesti tällaista tilannetta. (Akhmechet 2006)

```
string s1 = somewhatLongOperation1();
string s2 = somewhatLongOperation2();
string s3 = concatenate(s1, s2);
```

Kuva 28. Esimerkki tehtävästä, joka voidaan jakaa monen ytimen tehtäväksi

# 5 ESITELTYJEN OHJELMOINTIMALLIEN HYÖDYNTÄMINEN UNITYSSÄ

Dataorientoitunutta suunnittelua ja funktionaalista ohjelmointia ei voi, eikä myöskään kannata, yrittää käyttää sellaisenaan Unityssä. Sen sijaan näiden paradigmojen ominaisuuksia kannattaa hyödyntää ottaen huomioon niin Unity-moottorin arkkitehtuuri, kuin C#-kielen ominaisuudet ja rajoitteet. Paras lopputulos saadaan aikaiseksi arvioimalla projektin tarpeet ja käyttämällä mahdollisimman paljon hyödyksi niiden osa-alueiden vahvuuksia, joihin tarve on suurin. Tässä kappaleessa esitellään, miten esiteltyjä ohjelmointimalleja voi hyödyntää sekä yhdessä että erikseen Unityllä kehitettäessä. Esimerkkimallia kutsutaan tässä työssä Fudo-mallina, jonka nimi tulee funktionaalisesta ohjelmoinnista ja dataorientoituneesta suunnittelusta.

## 5.1 Logiikan ja tietorakenteiden erottaminen

Toteutetussa viitekehyksessä logiikka ja tietorakenteet on erotettu omiksi kokonaisuuksikseen. Tarkoituksena oli jakaa ohjelma pieniin kokonaisuuksiin, joita yhdistelemällä voidaan helposti ja nopeasti luoda uusia toiminnollisuuksia ohjelmaan. Pieniä kokonaisuuksia on helpompi hallita ja ymmärtää. Ohjelman rakenne on jaettu kolmeen yksikköön, kuten kuvassa 29 havainnoidaan: Entiteettiin, komponenttiin ja prosessoriin. Entiteetti on pelkkä ID-tunnistenumero, jolla samalla peliobjektilla olevat komponentit yhdistetään toisiinsa. Komponentti on luettelo tietorakenteista, joita prosessorit käyttävät. Prosessori vastaavasti sisältää logiikan, joka muuttaa komponenteissa olevaa tietoa.


Kuva 29. Tietorakenteen ja logiikan erottaminen

Prosessoreita hallinnoi prosessoriajuri, joka kutsuu prosessoreiden Update-, LateUpdate- ja FixedUpdate-funktioita. Ajuri määrittää prosessoreiden parametrit ja niiden suoritusjärjestyksen. Entiteettejä hallinnoi entiteettihallintaluokka, jonka avulla luodaan uusia entiteettejä ja poistetaan niitä käytöstä. Komponenteille on myös luotu oma hallintaluokkansa, jonne kaikki komponenttilistat on tallennettu, sekä niitä hallinnoivat funktiot. Prosessoriajuri, entiteettihallintaluokka ja komponenttihallintaluokka on tiivistetty seuraavassa kuvassa.


Kuva 30. Entiteettien, prosessorien ja komponenttien hallintajärjestelmät

Dataorientoituneen suunnittelun malli komponenteille on erittäin hyvä vaihtoehto toteuttaa ne Unityssä, vaikka se on paljon työläämpi rakentaa kuin Unityn oma arkkitehtuurimalli, jossa kaikki tieto on sidottu eri MonoBehaviour-luokista periytyviin komponenttiluokkiin. Toteuttamalla tietorakenteet DOD-mallin mukaisesti Unityssä saadaan kaikki edut siitä mukaan lukien tehokkaampi muistin käyttö ja komponenttien toisistaan riippumattomuus, joka kasvattaa modulaarisuutta. Dataorientoitunut suunnittelu yhdistettynä Unityn avoimuuteen tekee pelinkehitysympäristöstä käyttäjäystävällisen.

Pelilogiikan, eli tässä mallissa prosessoreiden, toteuttamisessa hyvä vaihtoehto on käyttää funktionaalisen ohjelmoinnin perusperiaatteita. Logiikoille annetaan vain ne komponentit, joiden tietoa muokataan funktion parametreissa. Funktioita kutsutaan prosessorihallintaluokasta, jolloin itse prosessorin ei tarvitse periä MonoBehaviour-luokkaa, joka toisi turhaa jätekuormaa roskienkeruujärjestelmälle. Yksittäisen asian suorittavia prosessoreita voidaan helposti käyttää rinnakkaislaskennassa, koska ulkoisista tiedoista riippumattomia prosessoreita voi siirtää helposti eri prosessorisäikeille ajettavaksi.

## 5.2 Komponenttien toteuttaminen Unityssä

Laajentamaton Unity toteuttaa sekä logiikan että tietorakenteet omissa MonoBehaviour-luokista periytyvistä luokissaan. Tämä arkkitehtuurinen ratkaisu on yksi Unityn suurimmista kompastuskivistä modulaarisuudelle. Sen sijaan on suotavampaa kirjoittaa kaikki tietorakenteet omiin komponentteihinsa, jotka eivät periydy MonoBehaviour-luokasta ja toteuttaa logiikka muualla. Dataorientoituneen suunnittelun malli komponenteille on yksi parhaista vaihtoehdoista toteuttaa ne Unityssä, vaikka se on työläintä rakentaa. Listoihin tallennetut tietorakenteet ovat muistiystävällinen valinta, ja koska mallin mukaisesti toteutetut tietorakenteet ovat toisistaan riippumattomia, on dataorientoitunut suunnittelu myös yksi modulaarisimmista valinnoista tietorakenteiden hallinnan toteuttamiseen.

Koska kaikki tieto tallennetaan listoihin, ei kaikkien eri tietorakenteiden ympärille tarvitse, eikä kannata, rakentaa omia komponenttejaan. Listoihin tallentaminen mahdollistaa pelkkää tyyppiä olevat komponentit, joita ei tarvitse erikseen luoda. Esimerkiksi tietoa siitä mikä on peliobjektin korkein mahdollinen nopeus, on järkevää hallita pelkässä float-muuttujassa, joka tallennetaan omaan float-listaansa.

Komponentteja suunnitellessa on tärkeää miettiä, mitä tietoa pelissä tarvitaan ja minkälainen tieto esiintyy usein yhdessä. Komponentit kannattaa rakentaa vain yhden tarkoitusperän ympärille, ja ne on hyvä pitää kompakteina, jotta niissä olisi mahdollisimman vähän ylimääräistä tietoa, jota ei tarvita niin usein. Kahdesta kuuteen muuttujaa on hyvä suuntaa antava määrä, mitä yhden komponentin pitäisi

sisältää. Jos yksi komponentti sisältää tätä enemmän muuttujia, on hyvä miettiä voiko komponentin jakaa kahtia järkevästi. Lisäksi Unitylle ohjelmoitaessa komponentit kannattaa tehdä sarjallistuvina, jotta ne ovat nähtävissä Unityn editorinäkymässä. Seuraavassa kuvassa on esimerkki yhdestä mahdollisesta komponentista Unityssä tehtävälle pelille.

```
[System.Serializable]
public class Movement {
    public Vector3 velocity;
    public Vector3 horizontalVelocity { get {
        return new Vector3(velocity.x, 0, velocity.z); } }
}
```

Kuva 31. Yksinkertainen sarjallistuva tietorakennekomponentti

Komponentteja luotaessa päätetään, tehdäänkö niistä luokkia vai tietueita. C++-kielessä nyrkkisääntö on, että pelkän tietorakenteen toteuttamiseen on hyvä käyttää tietueita, mutta Unityssä käytettävän C#-kielen tietueet eivät toimi kuten C++:ssa. Siinä missä C++:ssa tietueet eroavat luokista lähinnä vain sillä, että kaikki muuttujat ovat oletusarvoisesti julkisia, ero näiden kahden välillä C#:ssa on suurempi. Luokat ovat pelkästään viittauksia muistin osiin ja näin jokainen instanssi luokasta aiheuttaa vähän ylimääräisitä jätekuormaa roskienkeruulle. Tietueet sen sijaan ovat arvoja, ja näin ne siis ovat itse palanen muistissa.

Oletusarvoisesti itsetehtävät komponentit on hyvä luoda Unityssä luokilla tietueiden sijaan. Koska tietueet eivät aiheuta jätekuormaa on niiden luominen ja poistaminen luokkainstanssia nopeampaa. Luokat ovat kuitenkin helpompia käyttää, sillä listassa sijaitsevan tietueen muokkaaminen on mahdotonta. Sen sijaan, että tietueen informaatiota voitaisiin suoraan muuttaa, on siitä ensin tehtävä kopio, tehtävä muutokset kopioon ja sen jälkeen korvattava listan tietuekopiolla. Tämä on luokkatyyppistä muuttujan muuttamista työläämpi toteuttaa ja paljon hitaampaa. Tämän takia onkin hyvä arvioida, kuinka muuttuvaa komponentin tieto on ja sen perusteella valita, onko sen järkevä olla tietue vai luokka. Seuraavissa kuvissa on havainnoitu tietueen ja luokan muokkaamisen eroavaisuutta.

```
Vector3 newPosition = positions[index];
newPosition.y += verticalForce;
positions[index] = newPosition;
```

Kuva 32. Vector3-tietuelistassa olevan sijainnin muokkaaminen

```
positions[index].y += verticalForce;
```

Kuva 33. Vector3-luokkalistassa olevan sijainnin muokkaaminen

Unity pakottaa kaikki luotavat peliobjektit käyttämään niihin automaattisesti luotua Transform-komponenttia. Transform-komponentin muuttaminen on kuitenkin suhteellisen hidasta ja se sisältää ylimääräistä informaatiota monessa tilanteessa. Esimerkiksi sijaintia päivitettäessä ei ole tarvetta tiedolle Transform-komponentin rotaatiosta ja koosta. Näistä syistä on järkevää jakaa Transform-komponentti palasiksi, eli omiksi sijainti-, rotaatio-, ja kokokomponenteikseen, joita kaikki logiikat käyttävät laskuissaan. Sijainti ja koko ovat muotoa Vector3 ja rotaatio muotoa Quaternion, joten ne voi tallentaa sellaisinaan omiin listoihinsa. Jotta Unity saadaan reagoimaan esimerkiksi sijainnin muutoksiin, tarvitsee sitä varten vain luoda logiikka, joka sijoittaa kerralla kaikkien sijaintikomponenttien informaation Transform-listan komponentteihin.

Unity sisältää Transform-komponentin lisäksi myös muita valmiita komponenttiluokkia, esimerkiksi Rigidbody-luokan. Joitain näistä komponenteista on järkevää käyttää sellaisenaan, sillä Unity on juuriaan myöten optimoitu käyttämään pelimoottorista suoraan löytyviä komponentteja. Esimerkiksi kaikkien liikkuvien objektien on hyvä sisältää Rigidbody-komponentti, sillä kaikki Unityssä tapahtuva liikkuminen on optimoitu toimimaan parhaiten Rigidbody-komponentin kanssa. Transform-komponentin kaltaisesti muutkin Unity-komponentit sisältävät turhia ominaisuuksia ja ovat tietorakennekomponentteja raskaampia käyttää. Tämän vuoksi Unity-komponentteja kannattaa pitää vain loppusijoituspaikkoina, eikä tallentaa niihin mitään kesken laskujen. Tällainen ohjelmointityyli mahdollistaa myös sen, että mahdollisimman paljon pelin koodista toimii Unityn ulkopuolella, jolloin koodin siirtäminen esimerkiksi palvelimelle on helppoa. Myös Unity-komponentit listataan itsetehtyjen komponenttien tavoin, jotta niihin päästään helposti käsiksi. Objekteihin ne kuitenkin lisätään itsetehdyistä komponenteista poikkeavasti Unityn oletustavalla.

Kuten edellä on mainittu, listataan kaikki komponenttityypit omiin listoihinsa. Nämä listat on eroteltu toisistaan, eikä niiden toiminnallisuus ole millään tavalla riippuvainen toisistaan. Jotta listoja olisi mahdollisimman helppo käsitellä, on ne hyvä toteuttaa yhdessä paikassa, niitä varten luodussa komponenttihallintaluokassa. Kuten muut mahdolliset hallintaluokat, on komponenttihallintaluokan hyvä periytyä Singleton-luokasta. Tämä varmistaa, että komponenttihallintaluokista on vain yksi instanssi projektissa ja täten myöskään komponenttilistoista ei voi syntyä ylimääräisiä jäljennöksiä.

Komponenttilistat ovat järkevää toteuttaa Unityssä Dictionaryinä. Vaikka Dictionaryn läpi käyminen on hitaampaa kuin tavallisen taulukon, tai taulukkoon perustuvan Listin, mahdollistaa se komponenttien listaamisen epäjärjestyksessä. Toisin sanoen Dictionaryyn tallennettaessa ei ole väliä, millaisia arvoja entiteettien tunnisteet ovat, sillä siihen ei voi jäädä tyhjiä kohtia. List-taulukossa taas voi jäädä tyhjiä kohtia, eli jos esimerkiksi kohdassa kolme, eli entiteetillä, jonka tunniste arvo on kolme, ei ole komponenttia, on kohdassa silloin tyhjä viittaus, joka aiheuttaa ylimääräistä jätekuormaa roskienkeruulle. Lisäksi Dictionarystä on taulukkoja paljon nopeampi etsiä haluttua komponenttia, sillä Dictionaryssä hakunopeuteen ei vaikuta ollenkaan se kuinka paljon komponentteja joiden joukosta etsiä on, toisin kuin List-taulukossa, joka käy järjestyksessä kaikki arvonsa. Kaikkien komponenttilistojen avaintyyppi on kokonaislukumuuttuja. Avaimen virkaa listoille toimittaa entiteettinä toimiva indeksi, joka yhdistää samalla entiteetillä olevat komponentit toisiinsa. Koska listassa ei voi olla kahta samaa avainta samaan aikaa, varmistaa tämä sen, että samalle entiteetille ei voi kloonautua samaa tietoa koskaan.

Komponenttimanagerin sisällä olevat listatut komponenttityypit jaetaan neljään kategoriaan: Unity Component- ja Object-luokista periytyviin luokkiin, yksittäisiin tyyppiluokkiin, Unityn toteuttamiin luokkiin tai tietueisiin sekä käyttäjän itsensä luomiin komponentteihin. Nämä komponenttilistat on selvyyden vuoksi hyvä ryhmitellä komponenttihallintaluokassa tyyppiensä mukaisesti. Kuvassa 34 on esitelty erilaisia komponenttilistoja.

```
//Unityn komponentti-/objektilistat
public Dictionary<int, GameObject> entityGameObjects;
public Dictionary<int, Transform> entityTransforms;
public Dictionary<int, Rigidbody> rigidbodies;
//Muuttujatyyppilistat
public Dictionary<int, float> maxSpeeds;
public Dictionary<int, bool> isVisibles;
//Unity luokka/tietuerakennelistat
public Dictionary< int, Vector3> positions, scales, directions;
public Dictionary<int, Quaternion> rotations;
//Käyttäjän tekemistä komponenteista koostuvat listat
public Dictionary<int, Components.Controllable>
       controllableComponents;
public Dictionary< int, Components.Movement> movementComponents,
       previousFrameMovementComponents;
```

Kuva 34. Esimerkki komponenttilistoista komponenttihallintaluokassa

Komponenttilistojen hallintaa varten on järkevää luoda enumeraatio, joka määrittelee kaikille listoille, jotka eivät periydy Unity-Objectista tai -Componentista, oman komponenttityypin. Esimerkki tällaisesta enumeraatiosta on kuvassa 35. Komponenttihallintaluokka hyödyntää tätä enumeraatiota erilaisiin toimenpiteisiin, joissa pitää tietää samoista komponenttityypeistä oikea haluttu lista. Tällaisia funktioita ovat esimerkiksi komponentin piirtämistä Unity-editorissa varten luodut oikean komponentin asetus- ja palautusfunktiot, sekä halutun laisen komponentin lisääminen entiteettiin. Esimerkit näistä funktioista on esitelty kuvissa 36, 37 ja 38.

```
public enum ComponentType {
        Position,
        Scale,
        Direction,
        Rotation,
        MaxSpeed,
        IsVisible,
        Controllable,
        Movement,
        PreviousFrameMovement,
        Count
}
```

Kuva 35. Enumeraatio eri komponenttityypeistä

```
public void SetComponent(Enums.ComponentType componentType,
            Vector3 component, int entityId) {
   if(componentType == Enums.ComponentType.Position) {
      positions[entityId] = component;
   } else if (componentType == Enums.ComponentType.Direction) {
      directions[entityId] = component;
   } else if (componentType == Enums.ComponentType.Scale){
      scales[entityId] = component;
   } else {
       throw new ArgumentException("No component lists were found
                with the given type", "componentType");
   }
}
```

Kuva 36. Halutun komponentin asetusfunktio

```
public Vector3 ReturnVector3Component(Enums.ComponentType componentType,
       int entityId) {
  Vector3 vc;
  if (componentType == Enums.ComponentType.Position) {
     if (positions.TryGetValue(entityId, out vc)) {
         return vc;
     }
  } else if (componentType == Enums.ComponentType.Scale) {
     if (scales.TryGetValue(entityId, out vc)) {
         return vc;
     }
  } else if (componentType == Enums.ComponentType.Direction) {
     if (directions.TryGetValue(entityId, out vc)) {
         return vc;
     }
  } else {
    throw new ArgumentException("No component lists were found with the
          given type", "componentType");
  }
  Debug.LogWarning(string.Format("Entity doesn't have a component in the
          wanted component list, entity id: {0} component type: {1}",
          entityId, componentType));
  return Vector3.zero;
}
```

Kuva 37. Halutun komponentin palautusfunktio

```
public void AddComponent(Enums.ComponentType componentType,
     Vector3 component, int entityId) {
     if (componentType == Enums.ComponentType.Position) {
          positions.Add(entityId, component);
     } else if (componentType == Enums.ComponentType.Direction) {
          directions.Add(entityId, component);
     } else if (componentType == Enums.ComponentType.Scale) {
          scales.Add(entityId, component);
     } else {
          throw new ArgumentException("No component lists were
                 found with the given type", "componentType");
     }
          entityManager.entities[entityId].components.Add(
               componentType);
}
```

Kuva 38. Halutun tyyppisen komponentin lisäys haluttuun entiteettiin

Unity-Objectista ja -Componentista pohjautuville listoille on jo olemassa Unityn omat lisäys- ja poistofunktionsa. Näitä funktioita on järkevä käyttää kyseisille komponenteille, sillä ne ovat mahdollisimman optimoituja kyseisiä tarkoitusperiä varten. Omaa palautus- ja asetusfunktiotaan ne eivät myöskään tarvitse, sillä koodissa ei tarvitsee huolehtia tällaisten komponenttien piirtymisestä peliobjektissa, Unity hoitaa sen itse. Näistä syistä edellä esiteltyjä funktioita, sekä enumeraatiota ei tarvitsee toteuttaa Unityn omille Component- ja Object-luokille.

## 5.3 Entiteettien toteuttaminen Unityssä

Dataorientoituneella tiedon hallintatavalla poistetaan tietorakenteiden ja logiikan riippuvuus peliobjektien, olioiden, olemassaolosta. Sen sijaan, että nämä asiat olisi kahlittu toimimaan Unityn GameObject-komponentteina ne toimivat irrallisina omina kokonaisuuksinaan. GameObject-tyyppi ei kuitenkaan ole hyödytön, vaan jokaiseen pelimaailmassa esiintyvään entiteettiin voidaan linkittää oma GameObject-tyyppinsä. Listaamalla muiden komponenttien tapaisesti peliobjektit niiden hallinta helpottuu. Esimerkiksi tiettyä peliobjektia etsittäessä ei tarvitse käyttää hidasta GameObject.Find -funktiota, vaan se löydetään nopeasti peliobjektilistasta oikealla entiteettitunnisteella. Entiteetit, jotka eivät konkreettisesti esiinny pelimaailmassa, toimivat loistavasti ilman Unityn GameObject-tyyppiä. Tämä vähentää muistin rasitusta, sillä sekä peliobjektia, että Transform-komponenttia ei tarvitse luoda entiteettiä varten. Entiteettejä, joilla ei ole omaa peliobjektia voidaan käyttää

esimerkiksi erilaisina informaation tallennuspaikkoina, kuten tallennuspaikkojen osoittajana ja UI-grafiikkojen listaajina.

Unityn GameObject-peliobjekteihin, voidaan Unity-mallisesti linkittää niitä hallitsevat entiteetit. Peliobjektissa olevan entiteetin avulla saadaan näkyviin kaikki entiteetissä olevat komponentit. Tätä varten entiteetit tarvitsevat oman luokkansa, jonka pitää periytyä MonoBehaviourista. Entity-luokan luominen on järkevää, koska Unityn yksi vahvimmista puolista on tiedon näkyvyys ja muokkauksen helppous editorissa. Helppo tiedon muokattavuus helpottaa varsinkin suunnittelijoiden työtä pelinkehityksessä. Entiteettiluokka, josta on tehty esimerkki kuvassa 39, sisältää id-tunnisteensa lisäksi listan komponenttityypeistä, joita siihen on liitetty. Listan olemassaolo helpottaa entiteettien hallintaa. Esimerkiksi entiteettiä poistettaessa saadaan samalla helposti poistettua kaikki siihen linkitetyt komponentit oikeista listoista. Entiteetin poistofunktio on esitelty kuvassa 40.

```
public class Entity : UnityEngine.MonoBehaviour {
     public int id;
     public List<Enums.ComponentType> components;
}
```

Kuva 39. Entity-luokka

```
public void DeleteEntity(int id) {
    Entity entity;
    if(entities.TryGetValue(id, out entity)) {
       entities.Remove(id);
       foreach (Enums.ComponentType componentType in entity.components) {
          switch (componentType) {
            default:
              throw new System.Exception("Component deletion not implemented");

            case Enums.ComponentType.Controllable:
              componentManager.controllableComponents.Remove(id);
              break;
            case Enums.ComponentType.Direction:
              componentManager.directions.Remove(id);
              break;
            case Enums.ComponentType.IsVisible:
              componentManager.isVisibles.Remove(id);
              break;
            case Enums.ComponentType.MaxSpeed:
              componentManager.maxSpeeds.Remove(id);
              break;
            case Enums.ComponentType.Movement:
              componentManager.movementComponents.Remove(id);
              break;
            case Enums.ComponentType.Position:
              componentManager.positions.Remove(id);
              break;
            case Enums.ComponentType.Rotation:
              componentManager.rotations.Remove(id);
              break;
            case Enums.ComponentType.Scale:
              componentManager.scales.Remove(id);
              break;
          }
       }
       //Remove all references of Unity components and objects
       componentManager.entityTransforms.Remove(id);
       componentManager.rigidbodies.Remove(id);
       GameObject go;
       if (componentManager.entityGameObjects.TryGetValue(id, out go)) {
          Destroy(go);
          componentManager.entityGameObjects.Remove(id);
          Debug.Log("Deleted gameObject succesfully");
       }
       Debug.Log("Deleted entity succesfully");
    } else {
       Debug.Log("Did not found entity with this id");
    }
}
```

Kuva 40. Entiteetin poisto tunnisteen avulla

Entiteettien hallintaa varten on järkevä luoda oma hallintaluokkansa. Tämän luokan tehtävänä on pitää listaa pelissä olevista entiteeteistä. Kun entiteetit on tallennettu omaan Dictionary-listaansa, päästään halutulla entiteetin tunnisteella käsiksi sen entiteettiin ja sitä kautta saadaan informaatio, mitä komponentteja entiteetillä on. Tätä listaa voi hyödyksi käyttää entiteetin poistamiseen tunnisteavaimen avulla, mutta myös muihin toimintoihin, joissa pitää tietää mitä komponentteja entiteetillä on. Sen avulla voidaan myös etsiä haluttuja komponentteja, mutta tämä

mahdollisuus ei ole niin tarpeellinen, sillä komponentteja voi hakea suoraan komponenttilistoista halutulla indeksillä. Sen lisäksi että entiteettihallintaluokkaan on järkevä sijoittaa edellisen kuvan mukainen poistofunktio, kannattaa sinne sijoittaa esimerkiksi tapa hallita entiteettien tunnisteavaimia, jotta yhdelläkään entiteetillä ei voi olla samaa tunnisteavainta koskaan. Helpoin tapa toteuttaa tämä toiminto on juoksuttaa yhtä kokonaislukumuuttujaa isommaksi entiteettiä luotaessa, ja antaa uudelle entiteetille tunnisteeksi viimeisin luku. Esimerkki tällaisesta funktiosta on esitelty kuvassa 41.

```
public int GenerateEntityID() {
            do {
                if (!entities.ContainsKey(nextEntityID)) {
                    return nextEntityID;
                }
                nextEntityID++;
            } while (nextEntityID < maxEntites);
            throw new System.Exception("Entity capacity reached");
        }
```

Kuva 41. Uuden entiteetin tunnisteavaimen luonti

Entiteetin luontia ei ole järkevää toteuttaa entiteettihallintaluokassa, sillä erilaisten entiteettien mahdollisuus on käytännössä loputon, eikä sitä täten voi sitoa yhteen funktioon. Sen sijaan tarkoitusperästä riippuen entiteettien luonti voidaan tehdä käytännössä missä vain. Entiteetin voi luoda joko omassa hallintaluokassaan, mikä voi olla tärkeää joillekin entiteeteille, kuten pelaajalle. Sen lisäksi luontia varten voi luoda erilaisia varantoluokkia, joihin luodaan monta samanlaista entiteettiä nopeaa käyttöä varten. Viimeinen tapa on luoda entiteettejä normaalin koodin seassa ajon aikana. Entiteettiä luotaessa huomioitavat asiat ovat lähinnä mitä komponentteja entiteetti tulee tarvitsemaan. Näistä huomionarvoisin asia on määrittää, tarvitseeko entiteetti toimiakseen peliobjektia ja Unity-komponentteja vai voiko se toimia pelkällä puhtaalla tiedolla. Kuvassa 42. on esitelty pelaajahallintaluokassa tapahtuva pelaajaentiteetin luonti ja kuvassa 43. ajon aikana tapahtuva entiteetin luonti, joka ei tarvitse toimiakseen Unityn GameObject-tyyppiä.

```
public void CreatePlayer() {
     GameObject go = Instantiate(playerPrefab);
     Entity entity = go.GetComponent<Entity>();
     go.name = "Player";

     entity.id = entityManager.GenerateEntityID();
     entityManager.entities.Add(entity.id, entity);

     componentManager.entityGameObjects.Add(entity.id, go);
     componentManager.entityTransforms.Add(entity.id,
          go.transform);
     componentManager.rigidbodies.Add(entity.id,
          go.AddComponent<Rigidbody>());

     componentManager.AddComponent(Enums.ComponentType.Position,
          Vector3.zero, entity.id);
     componentManager.AddComponent(Enums.ComponentType.Rotation,
          Quaternion.identity, entity.id);
     componentManager.AddComponent(Enums.ComponentType.Direction,
          Vector3.zero, entity.id);
     componentManager.AddComponent(Enums.ComponentType.MaxSpeed,
          0, entity.id);
     componentManager.AddComponent(Enums.ComponentType.Movement,
          new Components.Movement(), entity.id);
     componentManager.AddComponent(
          Enums.ComponentType.Controllable,
          new Components.Controllable(), entity.id);
}
```

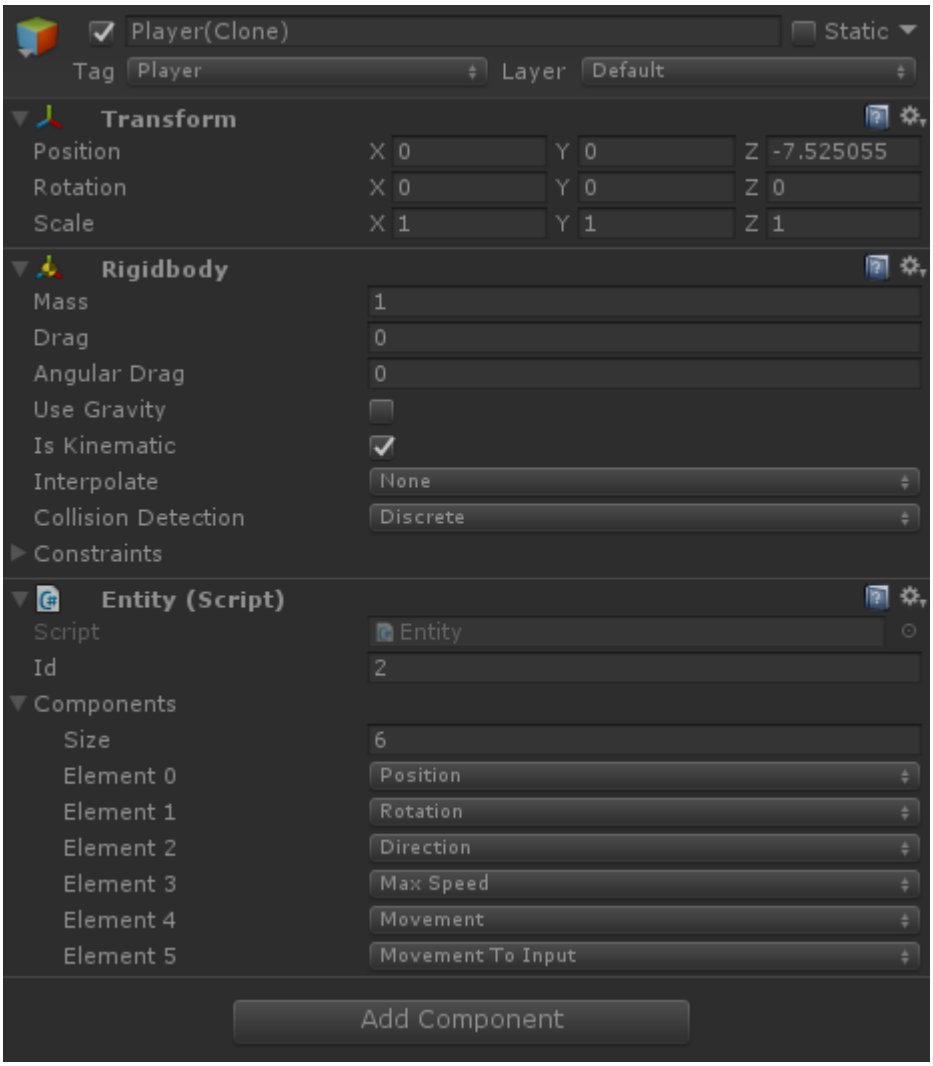
Kuva 42. Pelaajaentiteetin luonti

```
public void Update() {
  if (createCheckPointTimer >= createCheckPointInterval) {
      Entity entity = new Entity();
      entity.id = entityManager.GenerateEntityID();
      entityManager.entities.Add(entity.id, entity);

      componentManager.AddComponent(Enums.ComponentType.Position,
        componentManager.positions[player.id], entity.id);
      componentManager.AddComponent(Enums.ComponentType.Rotation,
        componentManager.rotations[player.id], entity.id);
      createCheckPointTimer = 0f;
  } else {
      createCheckPointTimer += Time.deltaTime;
  }
}
```

Kuva 43. Yksinkertaisen tallennuspaikkaentiteetin luonti pelin ajon aikana

Sen lisäksi, että entiteettiluokan olemassaolo helpottaa entiteettien hallintaa koodissa, se myös mahdollistaa entiteetin tietojen näyttämisen Unityn editorissa. Ilman lisämuokkauksia entiteettiin linkitetyt komponentit näkyvät seuraavan kuvan 44 kaltaisesti.




Kuva 44. Entity-luokan piirtyminen Unity-editorissa

Tällainen tiedon piirtyminen Unityssä ei ole suotavaa, sillä Unityn yksi pääkohdista ja vahvuuksista on tiedon vapaa näkyminen ja muokkaus editorissa. Tällaisessa versiossa käyttäjä näkee vain mitä komponentteja entiteetillä on, mutta ei sitä, mitä tietoa ne sisältävät. Tämän lisäksi käyttäjä voi poistaa ja lisätä listaan komponentteja, mutta tällainen toiminta ei poista komponentteja komponenttilistoista ollenkaan ja lisäksi ne säilyvät edelleen linkitettyinä entiteettiinsä.

Jotta peliobjektiinlinkitetty entiteetti saataisiin piirtämään sen omistamien komponenttien informaatio, täytyy sitä varten luoda CustomEditor-luokka. Tämä luokka hakee automaattisesti joka kehyksellä, kun sen entiteettiä tarkastellaan, informaatiota komponenttilistoista ja piirtää ne halutulla tavalla. Oikeiden komponenttien

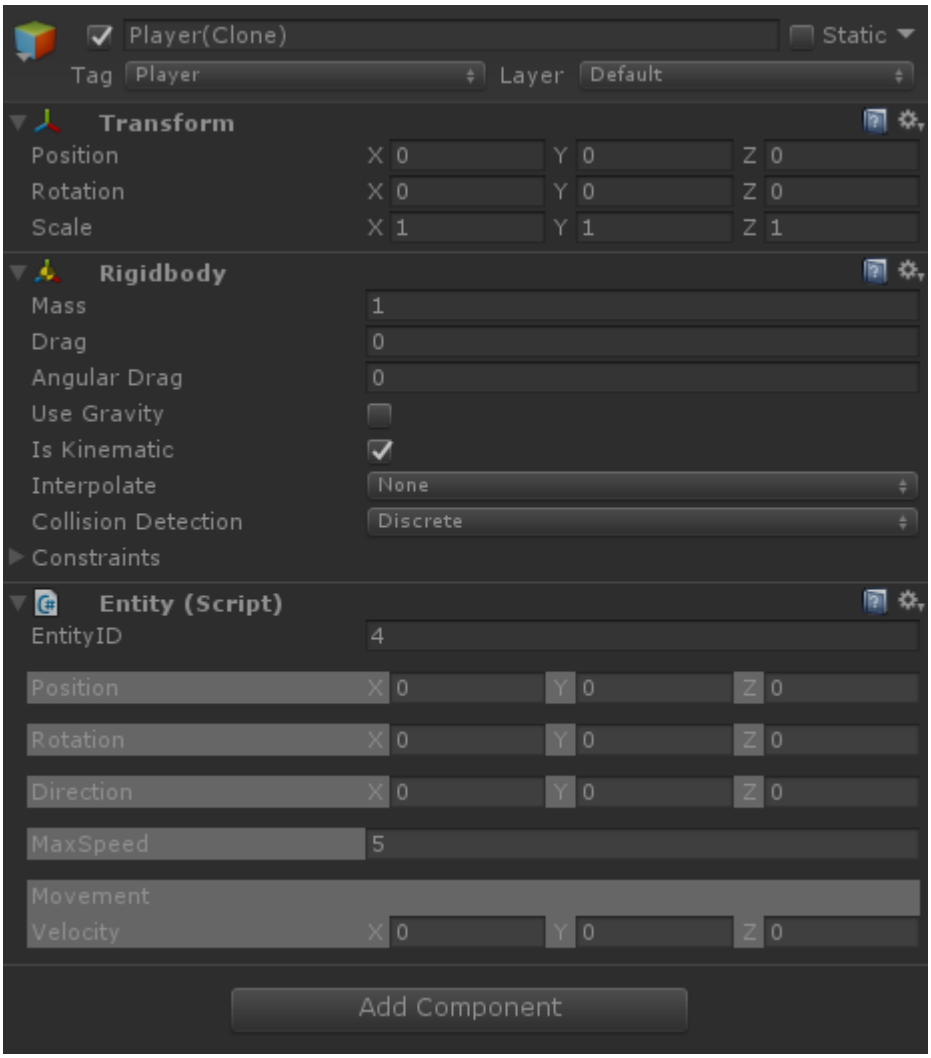
hakuun, CustomEditor-luokka käyttää entiteetin tunnistetta. Entiteetin komponenttilistaa taas käytetään optimoidusti siihen, että vain niistä komponenttilistoista haetaan informaatiota, missä entiteetti itse tietää sille kuuluvia komponentteja olevan. CustomEditor-luokka on Unity oma työkalu ja se mahdollistaa hyvin laajan määrän kustomoida tapoja, miten informaatiota näytetään editorissa. Tämän lisäksi CustomEditor-luokan avulla piirrettyä informaatiota voidaan muokata, jos muokattu tieto tallennetaan haetun tiedon päälle omassa rivissään. CustomEditor-luokan avulla voidaan myös tehdä napit komponenttien lisäystä ja poistoa varten, mikä lisää Unity-editorin hyötykäyttöä pelinkehityksessä. Kuvassa 45 on esitelty yksinkertainen koodi entiteetin piirtämiseen ja kuvassa 46. lopputulos.

```
[CustomEditor(typeof(Entity))]
 public class EntityEditor : Editor {
  public override void OnInspectorGUI() {
   Entity e = (Entity)target;
   EditorGUILayout.IntField("EntityID", e.id);

   if (ComponentManager.Instance == null) {
    return;
   }
   ComponentManager componentManager = ComponentManager.Instance;

   foreach (Enums.ComponentType componentType in e.components) {
    EditorGUILayout.Space();
    Rect rect = EditorGUILayout.BeginVertical();
    EditorGUI.DrawRect(rect, Color.gray);
    switch (componentType) {
      default:
       Debug.LogWarning("Component draw not implemented for: " +
                        componentType);
       break;
      case Enums.ComponentType.MaxSpeed:
       float f = componentManager.ReturnFloatComponent(componentType, e.id);
       f = EditorGUILayout.FloatField(componentType.ToString(), f);
       componentManager.SetComponent(componentType, f, e.id);
       break;
      case Enums.ComponentType.IsVisible:
       bool b = componentManager.ReturnBoolComponent(componentType, e.id);
       b = EditorGUILayout.Toggle(componentType.ToString(), b);
       componentManager.SetComponent(componentType, b, e.id);
       break;
      case Enums.ComponentType.Position:
      case Enums.ComponentType.Direction:
      case Enums.ComponentType.Scale:
       Vector3 v = componentManager.ReturnVector3Component
                   (componentType, e.id);
       v = EditorGUILayout.Vector3Field(componentType.ToString(), v);
       componentManager.SetComponent(componentType, v, e.id);
       break;
      case Enums.ComponentType.Rotation:
       Quaternion qt = componentManager.ReturnQuaternionComponent
                       (componentType, e.id);
       qt = Quaternion.Euler(EditorGUILayout.Vector3Field
            (componentType.ToString(), qt.eulerAngles));
       componentManager.SetComponent(componentType, qt, e.id);
       break;
      case Enums.ComponentType.Movement:
      case Enums.ComponentType.PreviousFrameMovement:
       EditorGUILayout.LabelField(componentType.ToString());
       Components.Movement mv = componentManager
       .ReturnMovementComponent(componentType, e.id);
       mv.velocity = EditorGUILayout.Vector3Field("Velocity", mv.velocity);
       break;
     }
     EditorGUILayout.EndVertical();
   }
  }
 }
```

Kuva 45. Yksinkertainen entiteetin piirtämisluokka




Kuva 46. Entiteetin piirto, kun entiteetin piirtämisluokka on käytössä

## 5.4 Pelilogiikan toteuttaminen Unityssä

Pelilogiikka on toteutettu Fudo-viitekehyksessä prosessori-luokissa. Prosessorissa on toteutus Unityn Update-, LateUpdate- ja FixedUpdate-funktioita vastaaville funktioille. Päivittäviä funktioita kutsutaan prosessoriajurista. Riippuen prosessorin tarkoituksesta käytetään joko yhtä tai kaikkia edellä mainittuja funktioita, jos niitä tahdotaan päivittää joka kehys. Prosessorit muokkaavat, funktionaalisen ohjelmointimallin mukaisesti, vain niitä tietoja, mitä funktioille annetaan parametreissa – eli viitekehyksen komponenttilistoja.

Prosessorit kannattaa mahdollisuuksien mukaan ohjelmoida niin, että mahdollisimman vähän ominaisuuksista on Unity-riippuvaisia. Itsenäisistä prosessoreista koostuvaa pelilogiikka on helpompi testata ja ohjelman suoritusjärjestystä vaivatonta muokata. Lisäksi, ohjelmointivirheitä tulee vähemmän, koska prosessorit käsittelevät vain tietorakenteita, jotka niille annetaan.

Viitekehyksen prosessorit on määritelty staattisina, jotta niitä voidaan käyttää mistä luokasta tahansa, ja koska ne toimivat samalla tavalla riippumatta siitä, mikä peliobjekti niitä käsittelee. Esimerkiksi objektin liikkuminen tapahtuu aina samoilla periaatteilla, joten jokainen voi käyttää samaa staattisista prosessoria, ainoastaan käytettävät komponentit vaihtuvat. Staattinen määrittely myös määrittää sen, että samaa prosessoria on kerralla ajossa vain yksi instanssi, joten samalla logiikalla ei muuteta samaa tietoa moninkertaisesti. Kun entiteetit voivat käyttää samoja prosessoria toimintoihinsa, modulaarisuus lisääntyy ja kehitys nopeutuu. Prosessoreita ei myöskään peritä MonoBehaviour-luokasta, joten ylimääräistä jätekuormaa jätteenkeruulle ei tule, varsinkaan kun Unityn omia päivitysfunktioita ei käytetä ollenkaan.

Toteutetussa viitekehyksessä käyttäjän syöte tallennetaan RawInput-prosessorilla, jossa käytetään Unityn Input-luokasta saatavia asetuksia. RawInput-prosessorin voi helposti korvata muissa pelienkehitystyökaluissa toimivaksi, koska Unitystä saatavat käyttäjän syötteet muutetaan viitekehyksistä riippumattomiksi enumeraatioiksi. Niitä käytetään myöhemmin esimerkiksi liikkumisen hallintaan MovementToInput-prosessorissa. Seuraavassa kuvassa on esitelty RawInput-prosessoriluokka Unityssä.

```
public static class RawInput {
     public static void Update(GenericDictionary<Components.BufferedInputs>
                               inputs) {
          foreach (KeyValuePair<int, Components.BufferedInputs> input in
                   inputs) {
               Components.BufferedInputs bufferedInputs;

               if (inputs.TryGetValue(input.Key, out bufferedInputs)) {
                    if (UnityEngine.Input.GetButtonDown("Forward")) {
                         bufferedInputs.inputs.Add(new
                         BufferedInput(Enums.Key.Up, Enums.KeyState.Down));
                    } else if (UnityEngine.Input.GetButtonUp("Forward")) {
                         bufferedInputs.inputs.Add(new
                         BufferedInput(Enums.Key.Up, Enums.KeyState.Up));
                    }
                    if (UnityEngine.Input.GetButtonDown("Backward")) {
                         bufferedInputs.inputs.Add(new
                         BufferedInput(Enums.Key.Down, Enums.KeyState.Down));
                    } else if (UnityEngine.Input.GetButtonUp("Backward")) {
                         bufferedInputs.inputs.Add(new
                         BufferedInput(Enums.Key.Down, Enums.KeyState.Up));
                    }
                    if (UnityEngine.Input.GetButtonDown("Left")) {
                         bufferedInputs.inputs.Add(new
                         BufferedInput(Enums.Key.Left, Enums.KeyState.Down));
                    } else if (UnityEngine.Input.GetButtonUp("Left")) {
                         bufferedInputs.inputs.Add(new
                         BufferedInput(Enums.Key.Left, Enums.KeyState.Up));
                    }
                    if (UnityEngine.Input.GetButtonDown("Right")) {
                         bufferedInputs.inputs.Add(new
                         BufferedInput(Enums.Key.Right, Enums.KeyState.Down));
                    } else if (UnityEngine.Input.GetButtonUp("Right")) {
                         bufferedInputs.inputs.Add(new
                         BufferedInput(Enums.Key.Right, Enums.KeyState.Up));
                    }
               }
          }
     }
}
```

Kuva 47. RawInput-prosessori

Näppäinten painalluksista luodaan suunta, johon hallittavaa entiteettiä liikutetaan InputToMovement-prosessorin avulla. Se saa tiedot käyttäjän näppäinten painalluksista ja suorittaa niiden muuttamisen kahteen ulottuvuuteen: Eteenpäin ja sivulle. Prosessorin toteuttaminen mahdollistaa sen, että käyttäjän antotietoja voitaisiin ottaa vastaan useamman kehyksen ajan ja tehdä sen perusteella muokkauksia prosessorin lopullisiin tietoihin. Kuvassa 48 on esimerkki InputToMovement-prosessoriluokasta.

```
public static class InputToMovement {
     public static void Update(GenericDictionary<Components.BufferedInputs>
                    inputs, GenericDictionary<Components.MovementInput
                    inputToMovements) {
          foreach (KeyValuePair<int, Components.BufferedInputs> input in inputs) {
               Components.MovementInput movementInput;
               if (inputToMovements.TryGetValue(input.Key, out movementInput)) {
                    for (int i = 0; i < input.Value.inputs.Count; i++) {
                         if (input.Value.inputs[i].key == Enums.Key.Up &&
                              input.Value.inputs[i].state == Enums.KeyState.Down) {
                              movementInput.forward += 1.0f;
                         }
                         if (input.Value.inputs[i].key == Enums.Key.Up &&
                              input.Value.inputs[i].state == Enums.KeyState.Up) {
                              movementInput.forward -= 1.0f;
                         }
                         if (input.Value.inputs[i].key == Enums.Key.Down &&
                              input.Value.inputs[i].state == Enums.KeyState.Down) {
                              movementInput.forward -= 1.0f;
                         }
                         if (input.Value.inputs[i].key == Enums.Key.Down &&
                              input.Value.inputs[i].state == Enums.KeyState.Up) {
                              movementInput.forward += 1.0f;
                         }
                         if (input.Value.inputs[i].key == Enums.Key.Left &&
                              input.Value.inputs[i].state == Enums.KeyState.Down) {
                              movementInput.right -= 1.0f;
                         }
                         if (input.Value.inputs[i].key == Enums.Key.Left &&
                              input.Value.inputs[i].state == Enums.KeyState.Up) {
                              movementInput.right += 1.0f;
                         }
                         if (input.Value.inputs[i].key == Enums.Key.Right &&
                              input.Value.inputs[i].state == Enums.KeyState.Down) {
                              movementInput.right += 1.0f;
                         }
                         if (input.Value.inputs[i].key == Enums.Key.Right &&
                              input.Value.inputs[i].state == Enums.KeyState.Up) {
                              movementInput.right -= 1.0f;
                         }
                         movementInput.forward =
                         Mathf.Clamp(movementInput.forward, -1.0f, 1.0f);
                         movementInput.right = Mathf.Clamp(movementInput.right, -
                         1.0f, 1.0f);
                    }
                    input.Value.inputs.Clear();
                    input.Value.inputs.Clear();
               }
          }
     }
}
```

Kuva 48. InputToMovement-prosessori

Esimerkiksi kuvassa 49 oleva CharacterMovement-prosessori laskee objektien seuraavan sijainnin käyttäjältä aiemmin tallennetun InputToMovement-tiedon perusteella. Prosessori käsittelee vain liikkumisarvoja sisältävää Movement-komponenteista koostuvaa listaa ja sijaintitietoja sisältävää Vector3-listaa. Uusi sijainti lasketaan lisäämällä nykyiseen sijaintiin nopeus. Huomattavaa on, että vasta funktion lopussa arvo asetetaan komponenttiin, näin komponentissa ei ole koskaan

niin sanotusti keskeneräistä arvoa tallennettuna. Tässä prosessorissa ei vielä aseteta sijainnin arvoa Unityn peliobjektiin, jotta prosessoria voitaisiin käyttää sen ulkopuolella helposti, esimerkiksi serverissä.

```
public static class CharacterMovement {
     public static void Update(GenericDictionary<Vector3> positions,
                               GenericDictionary<Components.Movement> movements,
                               GenericDictionary<Components.MovementInput> inputs) {
          foreach (KeyValuePair<int, Components.Movement>
                    movementComponent in movements) {
               Vector3 newPos;
               Components.MovementInput input;
               if (positions.TryGetValue(movementComponent.Key, out newPos)) {
                    if (inputs.TryGetValue(movementComponent.Key, out input)) {
                         Vector3 movePosition = newPos +
                              new Vector3(input.right, 0,
                         input.forward) * Time.deltaTime;
                         newPos = movePosition;
                         positions[movementComponent.Key] = newPos;
                    }
               }
          }
     }
}
```

Kuva 49. Esimerkki CharacterMovement-prosessorista

Kun Position-komponenttiin on laskettu seuraava sijainti, voidaan se asettaa peliobjektiin kuvan 50 esimerkin mukaisella prosessorilla UnityMovement. Se ottaa positions-komponentissa olevasta aiemmin lasketusta sijainnista uuden Vector3-arvon ja asettaa sen Rigidbody-komponenttiin MovePosition-funktion parametreissa. Prosessorissa on myös mahdollistettu peliobjektin siirtäminen antamalla arvo suoraan Transform-komponenttiin, jos peliobjektissa ei Rigidbody-komponenttia ole.

```
public static class UnityMovement
{
     public static void Update(Dictionary<int, Transform> transforms,
          Dictionary<int, Vector3> positions, Dictionary<int, Rigidbody>
          rigidbodies) {
          foreach (KeyValuePair<int, Vector3> position in positions) {
               Rigidbody rigidbody;
               if (rigidbodies.TryGetValue(position.Key, out rigidbody)) {
                    rigidbody.MovePosition(position.Value);
               } else {
                    transforms[position.Key].position = position.Value;
               }
          }
     }
}
```

Kuva 50. Esimerkki prosessorista, joka siirtää peliobjektia Unityssä

Prosessoriluokkien suoritusta ohjaa ProcessorManager-luokka, joka toimii pelilogiikka-ajurina. ProcessorManager periytyy Singleton<MonoBehaviour>-luokasta, jolloin se perii automaattisesti Unityn päivitysfunktiot, joista voidaan kutsua prosessoreiden päivitysfunktioita. Kuvan 51 esimerkissä oleva ProsessorManager-luokka ajaa ensin läpi pelaajan komennot vastaanottavan Input-prosessorin, jonka jälkeen se ajaa läpi liikkumisen laskevan Movement-prosessorin ja lopulta asettaa uudet arvot peliobjektiin UnityMovement-prosessorissa. Prosessoreiden suoritusjärjestys, ja niiden käyttämät komponentit, on helppo nähdä suoraan ProcessorManager-luokasta. Kun parametreissa on myös nähtävillä käytettävät komponenttilistat, prosessorit on helppo jakaa eri prosessoriytimille käytettäväksi, kunhan prosessorin käyttämä tieto ei ole muista riippuvaista.

```
public class ProcessorManager : Singleton<ProcessorManager> {
     protected ProcessorManager() { }
     private ComponentManager componentManager;

     public override void ReferenceManager() {
          componentManager = ComponentManager.Instance;
     }
     private void Update() {
          Processor.RawInput.Update(componentManager.playerInput);
          Processor.RawInputToAxis.Update(
               componentManager.controllableComponents,
               componentManager.playerInput);
          Processor.Input.Update();
          Processor.AxisToDirection.Update(
               componentManager.controllableComponents,
               componentManager.directions);
          Processor.Velocity.Update(componentManager.movementComponents,
               componentManager.directions, componentManager.rotations,
               componentManager.maxSpeeds);
          Processor.Movement.Update(componentManager.positions,
               componentManager.movementComponents);
          Processor.UnityMovement.Update(componentManager.entityTransforms,
               componentManager.positions, componentManager.rigidbodies);
     }
}
```

Kuva 51. Esimerkki ProcessorManager-logiikka-ajuriluokasta

Prosessorien sisäisissä laskuissa kannattaa käyttää puhtaita funktioita, joihin käytetään funktionaalista ohjelmointia. Luonnollisesti matematiikkakirjaston funktiot ovat puhtaita, koska ne palauttavat arvon annettujen parametrien perusteella. Projektiin on kannattavaa luoda matematiikkakirjasto, johon hyödyllisiä, usein käytettäviä, matemaattisia funktioita on hyvä koostaa. Kerralla oikein tehtyjen matemaattisten funktioiden käyttö vähentää ohjelmointivirheitä projektin edetessä. Kuvassa

52 on esitetty muutama esimerkkifunktio. IntersectionBetweenLines-funktio määrittää kahden janan välisen leikkauspisteen. Jos leikkauspistettä ei ole, funktio palauttaa Vector3.zero-arvon. Toinen kuvassa esitetty funktio ”AngleVectorPlane” laskee vektorin ja tason normaalin välisen kulman. Funktioihin voidaan myös piilottaa kiintolukuja. Esimerkiksi edellä mainitussa funktiossa, käytetään π / 2 -laskun likiarvoa.

```
public static Vector3 IntersectionBetweenLines(Vector3 linePoint1,
                        Vector3 lineVec1, Vector3
                        linePoint2, Vector3 lineVec2){
     Vector3 intersection;

     Vector3 lineVec3 = linePoint2 - linePoint1;
     Vector3 crossVec1and2 = Vector3.Cross(lineVec1, lineVec2);
     Vector3 crossVec3and2 = Vector3.Cross(lineVec3, lineVec2);

     float planarFactor = Vector3.Dot(lineVec3, crossVec1and2);

     // Tarkistaa, ovatko samalla suoralla
     if (Mathf.Abs(planarFactor) < 0.0001f &&
               crossVec1and2.sqrMagnitude > 0.0001f) {
          float s = Vector3.Dot(crossVec3and2, crossVec1and2) /
          crossVec1and2.sqrMagnitude;
          intersection = linePoint1 + (lineVec1 * s);
     } else {
          intersection = Vector3.zero;
     }
     return intersection;
}

public static float AngleVectorPlane(Vector3 vector, Vector3 normal){
     float dot;
     float angle;
     // Lasketaan pistetulo
     dot = Vector3.Dot(vector, normal);

     // Radiaaneina
     angle = (float)Math.Acos(dot);

     return 1.570796326794897f - angle;
}
```

Kuva 52. Matematiikkakirjastoon luotuja funktioita

Logiikkojen välinen viestintä hoidetaan tapahtumanhallintaluokan avulla. Se mahdollistaa tietojen ja tapahtumien lähettämisen ilman viittausten hakemista. Viittausten hakeminen voisi tuoda odottamattomia ongelmia ja ohjelman suoritusjärjestys täytyisi harkita tarkkaan, mikäli eri osat pelin logiikasta vaativat viittauksen johonkin tiettyyn luokkaan tai olioon ajon aikana.

Tapahtumahallintasysteemiä on helppo laajentaa lisäämällä uusia Dictionary-luokkia, joille annetaan tapahtuman tyyppi, jolla funktiota halutaan kutsua, ja annetaan Action-luokalle jokin tyyppi, kuten kuvassa 53 hahmotetaan.

```
public class EventManager : Singleton<EventManager>
{
     protected EventManager() { }

     private Dictionary<Enums.Event, Action> eventDictionary;
     private Dictionary<Enums.Event, Action<int[]>> intArrayEventDictionary;

     public override void Init() {
          eventDictionary = new Dictionary<Enums.Event, Action>();
          intArrayEventDictionary = new Dictionary<Enums.Event,
                                    Action<int[]>>();
     }
}
```

Kuva 53. EventManager-luokka, ja sen alustusfunktio

Tapahtumahallintaluokassa määritetään eri tietotyyppejä käyttävät tietorakenteet Dictionary-hakemistoon ja alustetaan ne tyhjinä. Ajon aikana hakemistoihin lisätään delegaatteja. Tapahtumahallintaluokasta kutsutaan tiettyjä tapahtumia Enums.Event-tyypillä, ja luokka kutsuu automaattisesti hakemistoon lisättyä delegaattia.

Jokaiselle eri arvoja sisältävälle tapahtumia varastoivalla tietorakenteelle luodaan tapahtumahallintaluokkaan oma StartListening-, StopListening- ja TriggerEvent-funktio, joita kutsutaan. Kun prosessorissa aloitetaan tai lopetetaan halutun tapahtuman kuuntelu, annetaan parametreissa tapahtuman tyyppi ja kuuntelijadelegaatti. Kuvissa 54 ja 55 havainnoidaan StartListening- ja StopListening- tapahtumanhallintafunktioita.

```
public static void StartListening(Enums.Event eventType, Action<int[]> listener)
{
     Action<int[]> thisEvent;
     if (instance.intArrayEventDictionary.TryGetValue(eventType, out thisEvent))
     {
          thisEvent += listener;
     } else {
          thisEvent += listener;
          instance.intArrayEventDictionary.Add(eventType, thisEvent);
     }
}
```

Kuva 54. Funktio tapahtuman kuuntelun aloittamiseen

```
public static void StopListening(Enums.Event eventType, Action<int[]> listener)
{
     if (eventManager == null) return;
     Action<int[]> thisEvent;
     if (instance.intArrayEventDictionary.TryGetValue(eventType, out thisEvent))
     {
          thisEvent -= listener;
     }
}
```

Kuva 55. Funktio tapahtuman kuuntelun lopettamiseen

TriggerEvent-funktiossa puolestaan annetaan parametreissa kutsuttavan tapahtuman tyyppi ja tapahtumatiedot, joka voi olla esimerkiksi kokonaisarvoja sisältävä kokonaislukutaulukko. TriggerEvent-funktiota voidaan kutsua kaikkialta ja kaikki prosessorit jotka ovat rekisteröityneet tapahtumatyypin kuuntelijoiksi vastaanottavat tapahtumassa saatavan informaation ja prosessoivat sen haluamallaan tavalla. Tapahtuman kutsuminen esitetään kuvassa 56.

```
public static void TriggerEvent(Enums.Event eventType, int[] eventData)
{
     Action<int[]> thisEvent = null;
     if (instance.intArrayEventDictionary.TryGetValue(eventType, out thisEvent))
     {
          thisEvent.Invoke(eventData);
     }
}
```

Kuva 56. Tapahtuman kutsuminen

## 5.5 Ohjelmointimallin tehokkuuden testituloksia

Esitellyn ohjelmointimallin (Fudo-malli) tehokkuus testattiin vertailemalla sitä Unityssä kahteen muuhun toteutustapaan: MonoBehaviour-luokilla tehtyyn ja LogicHandler-malliin, logiikka-ajurilla toimivaan kokonaisuuteen, jossa logiikka toteutettiin MonoBehaviour-luokista periytyvillä luokilla, joiden Update-funktiota kutsuttiin yhdestä logiikkojen hallintaluokasta. Tässä kappaleessa selvitetään, miten suuri vaikutus tietorakenteiden ja logiikan erotuksella on pelin suorituskykyyn.

Ero malleissa näkyy selvimmin tietorakenteiden käsittelystä. Fudo-mallissa pelilogiikkaa ajaa läpi vain yksi instanssi, joka käy listoja läpi ja muokkaa niiden tietorakenteita. MonoBehaviour-luokista koostuvassa toteutuksessa jokainen skripti lukee ja muokkaa omia tietojaan – eli toimintoja ei ajeta listoittain. LogicHandler-

malli on samankaltainen Fudo-mallin kanssa, mutta siinä peliobjektilla on oma entiteetti, johon tietorakenteet on tallennettu, ja joita logiikat muuttavat.

Unityn asetuksina käytettiin oletuksia, pois lukien Vsync-asetus, joka laitettiin pois päältä. Profilerissa testitulokset kerättiin Deep Profile -asetuksen ollessa päällä sekä pois päältä. Testituloksien keräämisen käytetty tietokone oli seuraavanlainen:

- Windows 10 Pro -käyttöjärjestelmä
- Intel i5-6600K -prosessori
- 16,0 Gb muistia

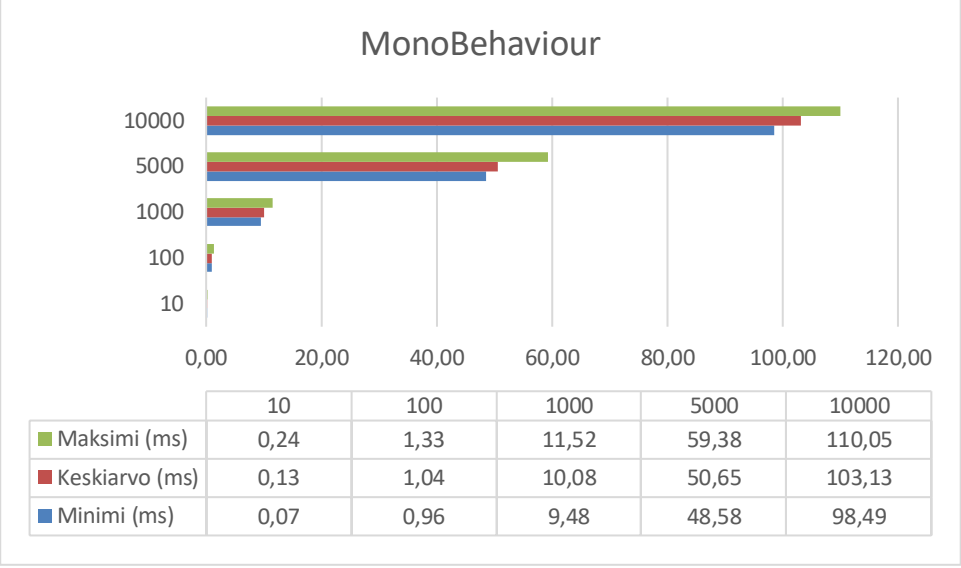
Täysin yhdenmukaisia logiikkoja ei pysty järjestelmien välillä tekemään, koska ne käsittelevät tietoa eri periaatteiden mukaisesti. Eri mallit pyrittiin kuitenkin toteuttamaan samalla tavalla, että testitulokset antaisivat mahdollisimman totuudenmukaisen kuvan eri mallien tehokkuuseroista. Liitteissä 1 ja 2 on kuvat eri Prefab-objekteista Unityn editorinäkymässä. Kaikilla suorituskykyyn vaikuttaa UnityEngine.Input-luokan syötteen tarkistaminen. Vaikutus on kaikille sama, joten keskinäinen vertailtavuus säilyy. Input-luokan vaikutus on myös todella pieni, sillä se tarkistetaan vain kerran kehyksessä, yhdessä logiikassa.

Tulokset saatiin käyttämällä Unitystä löytyvää Profiler-työkalua. Toimintamallien eroja testattiin minuutin mittaisella testijaksolla, jossa laskettiin peliobjektien nopeutta ja nopeuden mukaan määräytyvää sijaintia. Lisäksi, objektien luontia ja tuhoamista varten tehtiin omat testit. Objektien sijaintia laskevassa testissä odotettiin, että prosessorin kuorma tasoittui, jonka jälkeen testijakson mittaus aloitettiin. Testissä tehtiin otokset kymmenelle, sadalle, tuhannelle, viidelle tuhannelle ja kymmenelle tuhannelle objektille. Kymmenen objektia on pieni määrä, mutta se otettiin testiin sen takia, että on olemassa pelejä, joissa ei ole paljon liikkuvia objekteja. Luonti- ja tuhoamisprosessissa mittauskohtana käytettiin prosessorikuormassa tapahtuvia piikkejä. Kuormituspiikeistä otettiin viisi otosta, joista laskettiin keskiarvo. Luontiprosessista tehtiin kaksi erilaista testausta, joista toisessa peliobjekteihin lisättiin Unityn AddComponent-funktiolla tarvittavat tietorakenteet ja logiikat ja toisessa käytettiin valmiita Prefab-objekteja. Minimi-, maksimi- ja keskiarvojen laskemiseen käytettiin Github-käyttäjä Steve3003:n kehittämää Unity Profiler

Data Exporter -skriptikokoelmaa. (Saatavilla: https://github.com/steve3003/unity-profiler-data-exporter)

Aluksi testit tehtiin Unityn Profiler-työkalun Deep Profile -asetus päällä, jonka perusteella tulokset yllättävästi näyttivät, että MonoBehaviour-toteutus olisi ollut muita malleja jopa kaksinkertaisesti tehokkaampi. Suorituskykyerot johtuivat kuitenkin siitä, että Deep Profile –asetus tallentaa kaikki funktiokutsut, myös tietorakenteiden sisällä tapahtuvat. Tämä aiheuttaa sen, että eri tavalla toimivat mallit eivät ole vertailukelpoisia keskenään Deep Profile -asetus päällä, koska mallit, joissa käytetään enemmän funktioiden sisäisiä kutsuja, vievät asetus päällä enemmän tehoa ja tuottavat paljon ylimääräistä jätekuormaa – vaikka todellisuudessa mallit saattaisivatkin olla tehokkaampia. Tämän vuoksi vertailutestit tehtiin ilman Deep Profile -asetusta, liitteessä 3 on kuvattu alkuperäiset testitulokset havainnoimaan tehoeroja, mitä Deep Profile aiheuttaa.

Ohjelmointimallien tehokkuuden tarkastelu aloitettiin MonoBehaviour-luokilla toteutetuista logiikoista. Aluksi haluttiin tarkastella mallia, jonkalaista yleensä ohjelmoija Unityssä ensimmäiseksi tekee. MonoBehaviour-toteutuksesta saatiin pohja, johon muita toteutustapoja on hyvä verrata. MonoBehaviour-logiikkojen toteuttaminen on projektin alussa nopeaa, koska riippuvuuksia ohjelman eri osien välillä ei ole. Kuvassa 57 on tarkastelussa MonoBehaviour-luokilla tehdyn toteutuksen ajon aikainen suorituskyky.

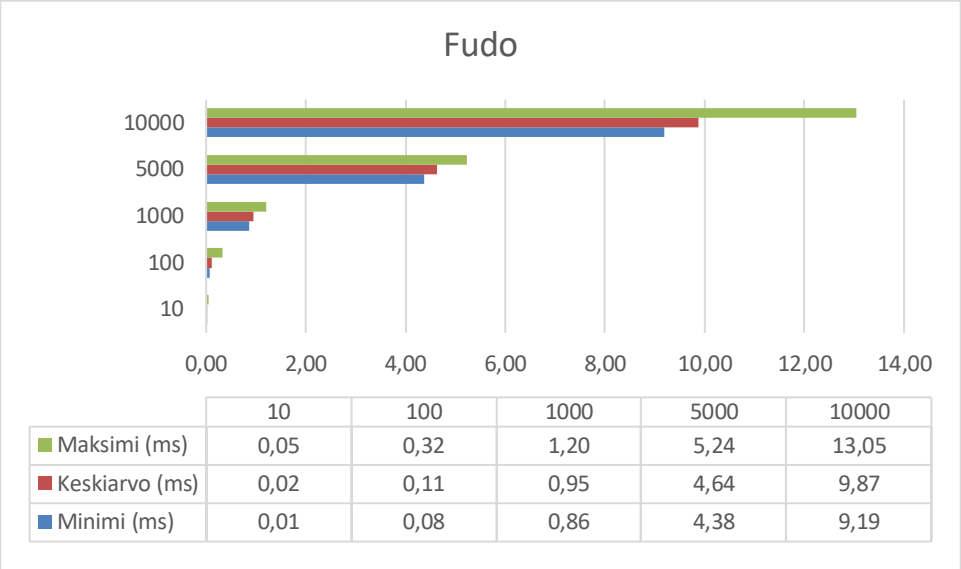



| | 10 | 100 | 1000 | 5000 | 10000 |
|---|---|---|---|---|---|
| Maksimi (ms) | 0,24 | 1,33 | 11,52 | 59,38 | 110,05 |
| Keskiarvo (ms) | 0,13 | 1,04 | 10,08 | 50,65 | 103,13 |
| Minimi (ms) | 0,07 | 0,96 | 9,48 | 48,58 | 98,49 |

Kuva 57. MonoBehaviour-toteutuksen suorituskykymittausten tulokset

Kuten kuvasta 57 nähdään, suorituskyky laskee lineaarisesti. Mitä useampia peliobjekteja, ja niiden sijaintia sekä nopeutta, käsitellään ajon aikana, sitä hitaampi MonoBehaviour-toteutus on. Jokaisesta päivitettävästä objektista syntyy ylimääräistä jätekuormaa joka kehys, mikä aiheuttaa sen, että ylimääräinen peliobjektien päälle tuleva tehonkulutus nousee sitä enemmän mitä enemmän peliobjekteja on. Jotta saavutetaan 60 ruutua per sekunti -päivityssykli, pitää käsittelyajan pysyä alle 16,67 ms per kehys. MonoBehaviour-mallin toteutuksella pelin logiikasta tulee testikoneen pelikäyttöön liian hidas jo viiden tuhannen objektin kohdalla.

Seuraava tutkittu malli oli aiemmissa aliluvuissa esitelty Fudo-malli. kuten muissakin malleissa, tässäkin tutkittiin minimi-, maksimi- ja keskiarvot eri objektimäärillä. Kuvassa 58 on esitetty Fudo-mallin suorituskykymittaukset.

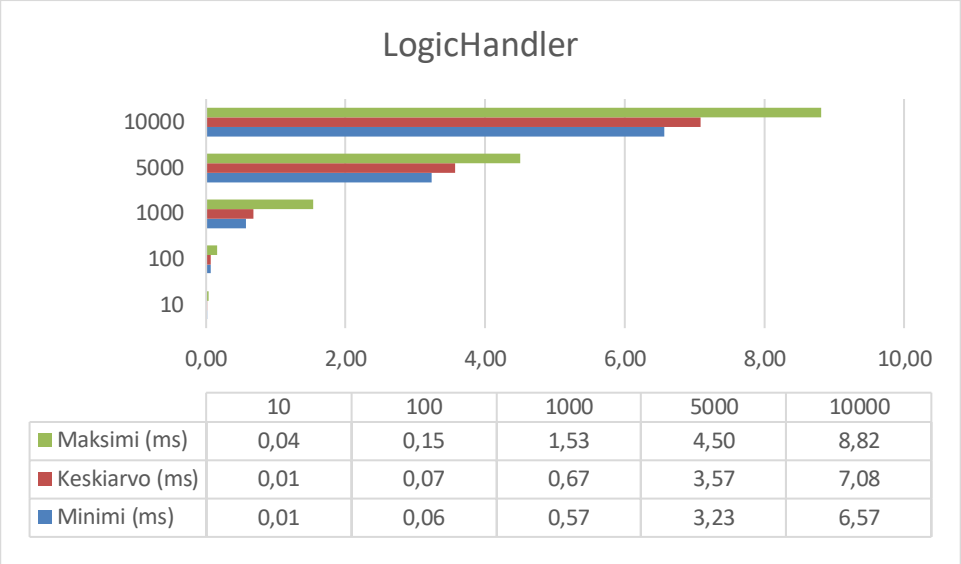



| | 10 | 100 | 1000 | 5000 | 10000 |
|---|---|---|---|---|---|
| Maksimi (ms) | 0,05 | 0,32 | 1,20 | 5,24 | 13,05 |
| Keskiarvo (ms) | 0,02 | 0,11 | 0,95 | 4,64 | 9,87 |
| Minimi (ms) | 0,01 | 0,08 | 0,86 | 4,38 | 9,19 |

Kuva 58. Fudo-viitekehyksen suorituskykymittausten tulokset

Fudo-mallin testauksessa havaittiin, että myös tässä mallissa suorituskyky laskee lineaarisesti, sillä erotuksella, että teho pikemminkin laskee vähän hitaammin kuin objektien määrän moninkertaistuminen. Fudo-mallin toteutuksessa suorituskyky pysyy erinomaisena, sillä vaikka päivitettäviä objekteja olisi kymmenen tuhatta, pysyy pelin ruudunpäivitysnopeus yli kuudessa kymmenessä testikoneella. Fudo-malli tuo myös vakautta suorituskyvylle, sillä vaihteluväli suorituksessa on määrällisesti tasaisempi verrattuna MonoBehaviour-malliin – vaikkakin kymmenen tuhannen objektin testissä se on prosentuaalisesti suurempi.

Viimeinen tarkasteltava malli oli logiikka-ajurilla, testauksia varten luotu, kokonaisuus. Se on niin sanotusti välimalli MonoBehaviour- ja Fudo-malleista. LogicHandler-malli lisättiin, koska haluttiin selvittää, että minkälainen suorituskykyero on MonoBehaviour-luokan Update-funktion ja oman vastaavan luokkien Update-funktioita kutsuvan logiikka-ajurin välillä. Luomalla tämä malli haluttiin myös saada tarkempaa tietoa, onko monen logiikan listassa päivittäminen nopeampaa, kuin listojen päivittäminen yhdessä logiikassa. Tulokset ovat nähtävissä kuvassa 59.

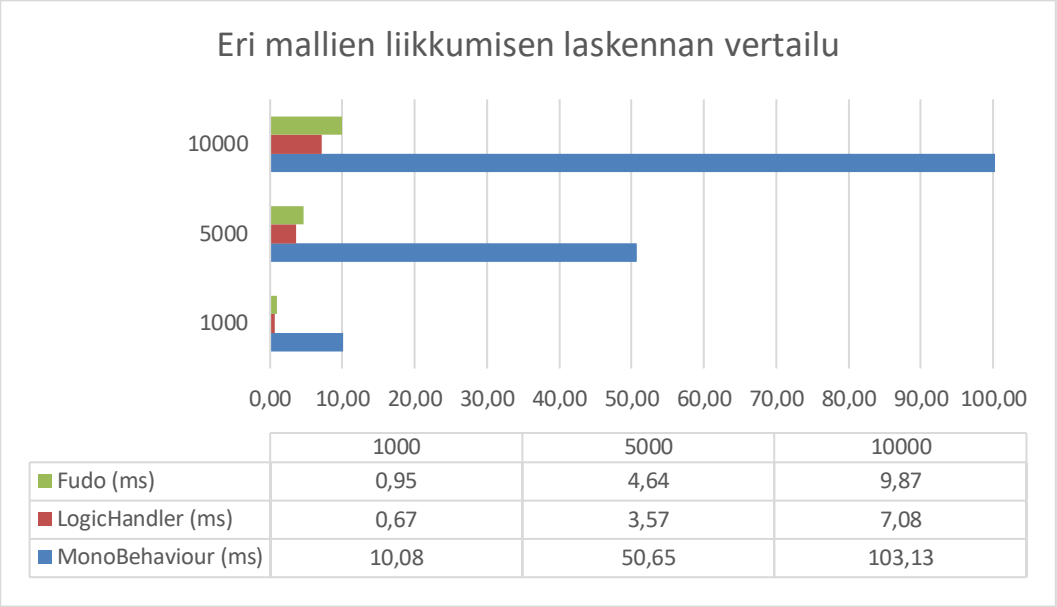



| | 10 | 100 | 1000 | 5000 | 10000 |
|---|---|---|---|---|---|
| Maksimi (ms) | 0,04 | 0,15 | 1,53 | 4,50 | 8,82 |
| Keskiarvo (ms) | 0,01 | 0,07 | 0,67 | 3,57 | 7,08 |
| Minimi (ms) | 0,01 | 0,06 | 0,57 | 3,23 | 6,57 |

Kuva 59. LogicHandler-toteutuksen suorituskykymittausten tulokset

LogicHandler-mallin testitulokset olivat hyvin samankaltaiset edellisten tulosten kanssa niin lineaarisesti laskevassa suorituskyvyn kuin pienen vaihteluvälinkin osalta. Fudo-mallin tapaisesti myös LogicHandler-malli säilytti erinomaisen suorituskyvyn vielä kymmenen tuhannenkin objektin kohdalla.

Kuvassa 60 havainnoidaan eri mallien suorituskykyeroja tuhannen, viiden tuhannen ja kymmenen tuhannen objektin välillä. Kymmenen ja sadan objektin testejä ei otettu mukaan keskinäiseen vertailuun, sillä niiden tehonkulutus oli vähäinen. Vertailuarvoiksi otettiin kustakin määrästä keskiarvot, jolloin saatiin todenmukaisin suorituskyky selville.

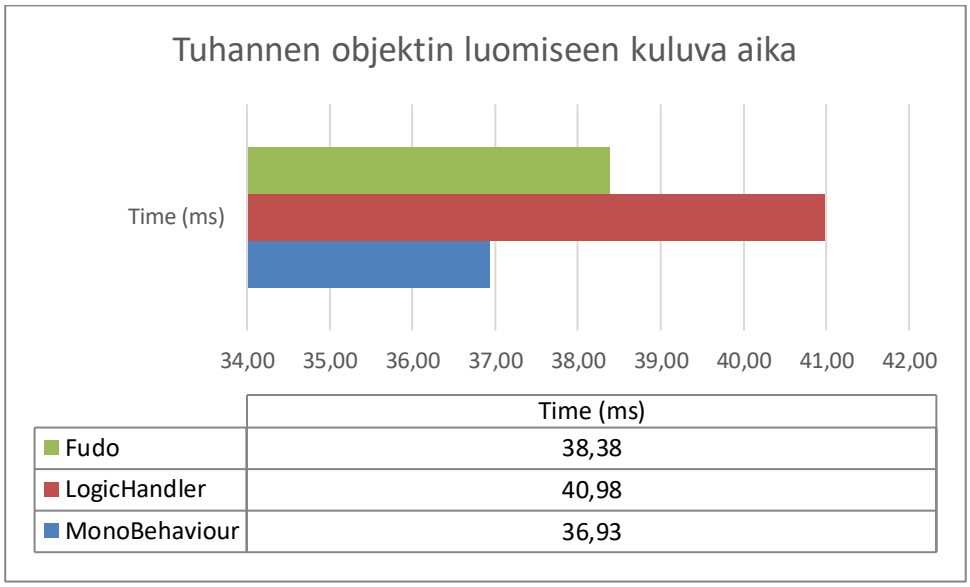



| | 1000 | 5000 | 10000 |
|---|---|---|---|
| Fudo (ms) | 0,95 | 4,64 | 9,87 |
| LogicHandler (ms) | 0,67 | 3,57 | 7,08 |
| MonoBehaviour (ms) | 10,08 | 50,65 | 103,13 |

Kuva 60. Eri mallien keskinäiset suorituskykyerot

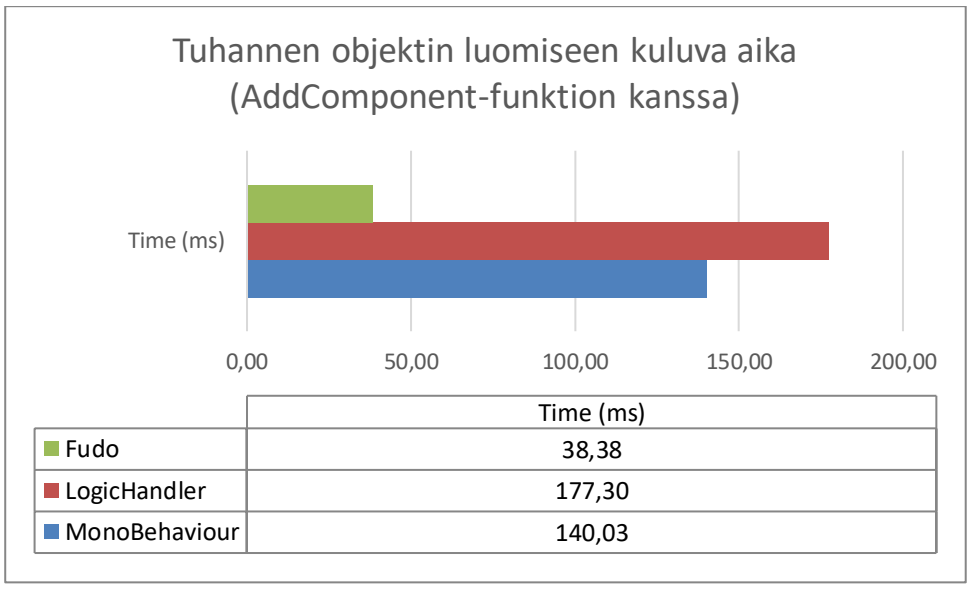
Testituloksia vertailtaessa havaittiin, että LogicHandler-mallin toteutus oli hivenen nopeampi kuin Fudo-malli – kummatkin mallit olivat kuitenkin selkeästi MonoBehaviour-toteutusta nopeampia. LogicHandler-malli on Fudo-mallia nopeampi, koska se käyttää tietorakenteiden käsittelyyn List-tietorakenteita, kun taas Fudo-malli käyttää Dictionary-tietorakenteita, joissa iterointi vie enemmän aikaa kuin List-tietorakenteissa. Toinen syy sille, että Fudo-malli on LogicHandler-mallia hitaampi, on modulaarisuus. Fudo-mallin logiikoissa käsitellään useampia listoja kuin LogicHandler-mallissa, joka johtaa siihen, että listojen käsittely tuo enemmän prosessorikuormaa Fudo-mallissa kuin LogicHandler-toteutuksessa. Fudo-mallia pystyy helposti optimoimaan yhdistelemällä tietorakenteita samoihin komponenttilistoihin, joka vähentää modulaarisuutta ja tehottomampi tapa käyttää muistia.

Seuraavaksi testattiin tuhannen objektin luomiseen kuluvaa aikaa. Objektit luotiin kerralla, joten suuri osa prosessorikuormasta tulee Unityn Instantiate-funktion käytöstä. Kuvassa 61 ja 62 on esitelty vertailun tulokset.




| | Time (ms) |
|---|---|
| Fudo | 38,38 |
| LogicHandler | 40,98 |
| MonoBehaviour | 36,93 |

Kuva 61. Tuhannen objektin luomiseen kuluva aika



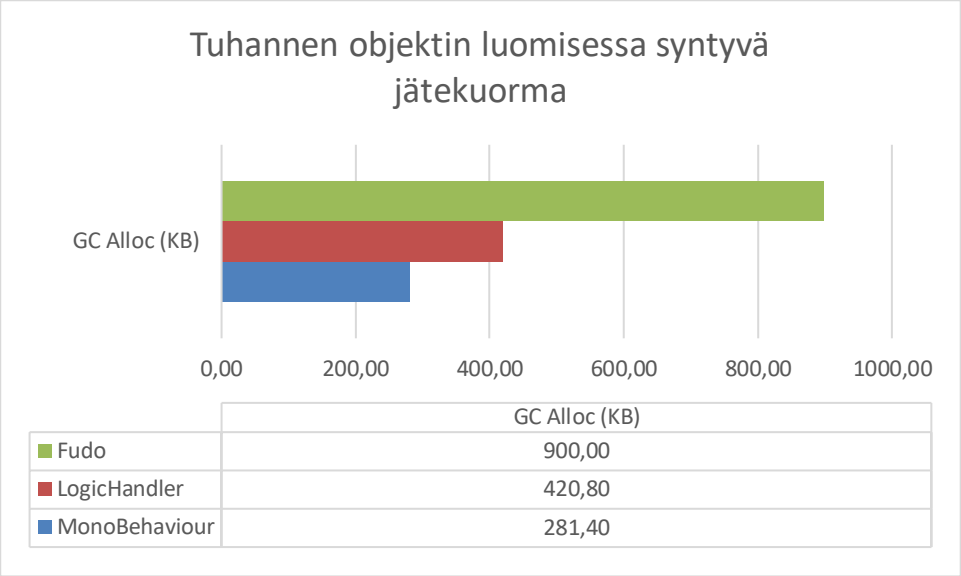

| | Time (ms) |
|---|---|
| Fudo | 38,38 |
| LogicHandler | 177,30 |
| MonoBehaviour | 140,03 |

Kuva 62. Tuhannen objektin luomiseen kuluva aika, kun komponentit lisätään luomisen yhteydessä

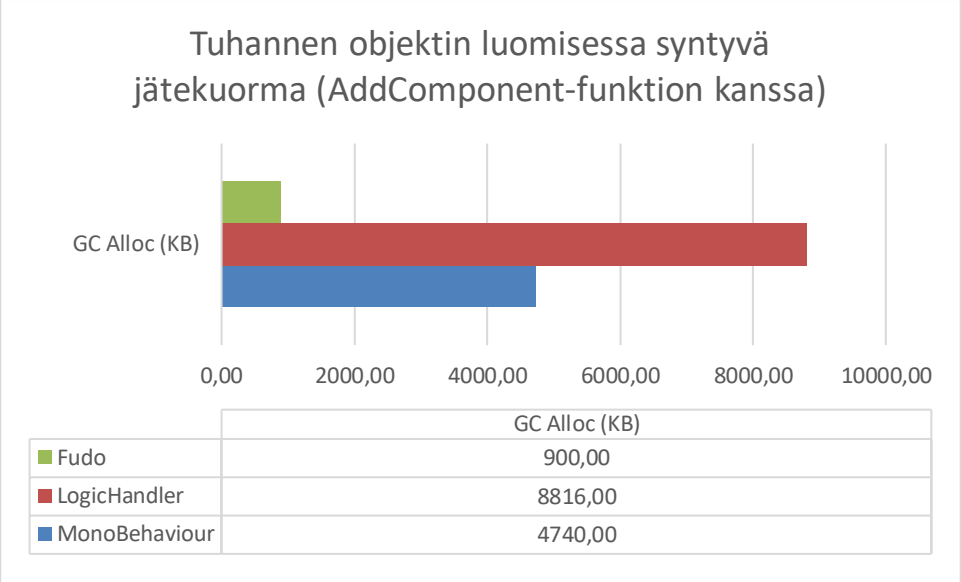
Kuvasta 61 huomattiin, että MonoBehaviour-malli oli testatuista nopein, erot olivat kuitenkin pienet: Hitain malli oli noin 10 % hitaampi kuin nopein malli. Fudo-mallin suorituskyvyn rasitteena verrattuna muihin malleihin on se, että siinä ei käytetä valmiita logiikkoja ja tietorakenteita sisältävää Prefab-objektia, vaan peliobjektiin lisätään tietorakennekomponentit ajon aikana. LogicHandler-mallin hitaus verrattuna MonoBehaviour-toteutukseen tulee siitä, että siinä logiikkojen on lisättävä itsensä logiikka-ajurin listaan, tämä vaatii ylimääräistä laskenta-aikaa. Kuvasta 62

nähdään erot, kun kaikkiin malleihin lisätään komponentit ajon aikana. Fudo-malli on yli kolme kertaa nopeampi kuin seuraavaksi nopein malli.

Objektien luontiin liittyen mitattiin myös sitä, miten roskienkeruu varaa muistia objekteita luotaessa. Syntynyt jätekuorma esitetään kuvassa 63 ja 64.




Kuva 63. Tuhannen objektin luomisessa syntyvä jätekuorma

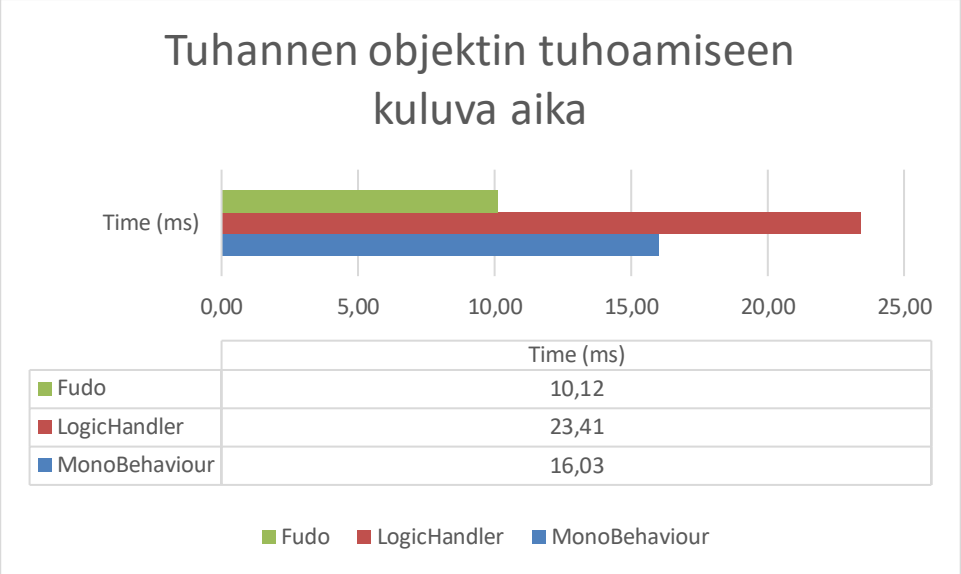



Kuva 64. Tuhannen objektin luomisessa syntyvä jätekuorma, kun komponentit lisätään luomisen yhteydessä

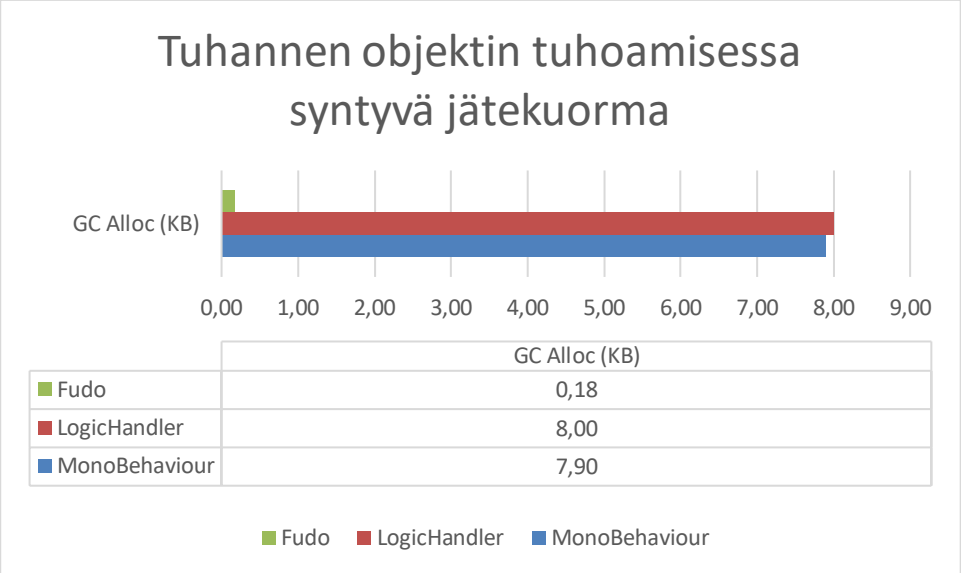
Kuvasta 63 nähdään, että Fudo-malli synnyttää eniten jätekuormaa. Tämä johtuu siitä, että Fudo-mallissa lisätään tietorakenteita peliobjekteihin ja asetetaan viit-

tauksia niihin. Sen lisäksi, että komponenttilistoille luominen ja lisääminen tuottavat jätettä, entiteetin piirtoa varten entiteetin komponenttilistaan lisääminen synnyttää ylimääräistä muistinvarausta. Kuvasta 64 nähdään erot, kun LogicHandler- ja MonoBehaviour-malleissa komponentit lisätään ajon aikana Fudo-mallin tapaisesti. Tällöin Fudo-malli on selkeästi kevein testatuista malleista.

Niin ikään objektien tuhoamiseen kuluvaa aikaa ja syntyvää jätekuormaa testattiin. Eri malleilla on hieman eroavat toteutukset siinä, miten objektit poistetaan käytöstä, jota on havainnollistettu kuvissa 65 ja 66.




Kuva 65. Tuhannen objektin tuhoamiseen kuluva aika




Kuva 66. Tuhannen objektin tuhoamisessa syntyvä jätekuorma

Fudo-malli on objektin poistamisessa nopein, eikä se aiheuta roskienkeruujärjestelmälle jätekuormaa. Muilla tavoilla jätekuormaa synnyttää GameObject.FindObjectsWithTag-funktion käyttö, sillä niitä varten ei ole luotu minkäänlaista listaa, jonka avulla ne voisi kaikki helposti poistaa. Toisaalta, FindObjectsWithTag-funktion käyttö voi olla perusteltua, sillä ilman sitä tarvittaisiin samantyylinen objektienhallintajärjestelmä, mitä Fudo-mallissa on, joka taas lisää luomisprosessissa syntyvää prosessorikuormaa. LogicHandler-luokalla tehty toteutus on MonoBehaviour-mallia hitaampi, koska sen täytyy poistaa viittaukset logiikka-ajurista.

Tarkemmat testitulokset ovat nähtävissä liitteissä 4 – 7. Yleistä Unityn jätekuormaa ei vertailtu, koska sen merkitys suorituskyvylle oli jokaisessa mallissa merkityksetön: Alle yksi millisekunti 10 000 objektilla ajon aikana.

Testitulosten perusteella jo pelkän logiikka-ajuriluokan luominen on kannattavaa, koska se on ajossa huomattavasti MonoBehaviour-mallia nopeampi – vaikka luonti- ja tuhoamisprosessit ovat hitaampia. Logiikka-ajurissa on vain yksi MonoBehaviour-kutsuja vastaanottava luokka, joka kutsuu omia päivitysfunktioita, jolloin funktiokutsujen seuraamisesta syntyvää jätekuormaa ei synny samalla tavalla kuin puhtaasti MonoBehaviour-luokista koostuvassa toteutuksessa. Logiikka-ajurimalli mahdollistaa myös sen, että logiikkojen suoritusjärjestystä voi helposti muuttaa.

Vaikka Fudo-malli ei olekaan kaikissa tilanteissa prosessorikuormaa tarkastellessa tehokkain, on sen muistinkäyttö tehokkaampaa kuin muissa malleissa. Lisäksi, Fudo-mallilla pelilogiikan ohjelmoiminen on helpompaa, koska osat ovat modulaarisia sekä objektien lisääminen ja poistaminen pelissä on yksinkertaista ja nopeaa.

Fudo-malli on optimointivaiheessa paljon parempi, koska itsenäisesti osista koostuvia, ja samoja komponentti-Dictionaryjä käyttäviä, logiikkoja on helpompi yhdistää – jolloin Dictionary-hakuja tulee vähemmän ja ohjelman suorituskyky paranee. MonoBehaviour-logiikkoja yhdistäessä pitää tehdä uusiksi viittaushaut ja kirjoittaa jopa logiikkaa uusiksi. Lisäksi pitää huolehtia, että logiikkoja käyttävä peliobjekti päivitetään käyttämään uusia oikeita logiikkoja. Fudo-mallissa peliobjektia ei tarvitse päivittää millään tavalla ja sen etu on siinä, että Prefab-objektissa ei tarvitse

muistaa laittaa montaa logiikkakomponenttia kiinni. Tämä vähentää virhealttiutta ja riippuvuutta, joka syntyy, kun logiikkakomponentti ei löydä peliobjektista toista logiikkakomponenttia, jota se tarvitsee toimiakseen.

Fudo-mallilla tehtäessä on tärkeää löytää tasapaino prosessorin suorituskyvyn ja muistin suorituskyvyn välillä. Muistille optimoidut yhden tyypin listat, jotka ovat luonnollisesti myös modulaarisempia käyttää ovat hitaampia prosessorille Dictionary.TryGetOut-funktiokutsun hitauden vuoksi. Esimerkkiprojektissa suosittiin yhden tyypin komponenttilistoja, niiden modulaarisuuden ja helppolukuisuuden vuoksi.

## 5.6 Viitekehyksiä Unitylle

Kuten edellisten kappaleiden esimerkeistä näkee, Unityn arkkitehtuurin parantaminen ja nykyaikaistaminen on suurehko työ. Tämä voi olla monelle kehittäjälle liian vaativa ja aikaa vievä prosessi, jolloin sen toteuttaminen ei ole kannattavaa. Useat tahot ovat kehittäneet omia viitekehyspakettejaan Unityn arkkitehtuurin korjaamiseen, ja niitä on jaossa ilmaiseksi Internetissä. Moni näistä paketeista toteuttaa entiteettikomponenttisysteemimallia, joka noudattaa lähinnä dataorientoituneen suunnittelun perusperiaatteita, mutta osassa on hyödynnetty myös funktionaalista ohjelmointia. Näiden kehyspakettien käyttö on ihanteellinen vaihtoehto Unity-kehittäjälle, jolla ei ole tarpeeksi aikaa ja resursseja oman viitekehyspakettinsa luontiin.

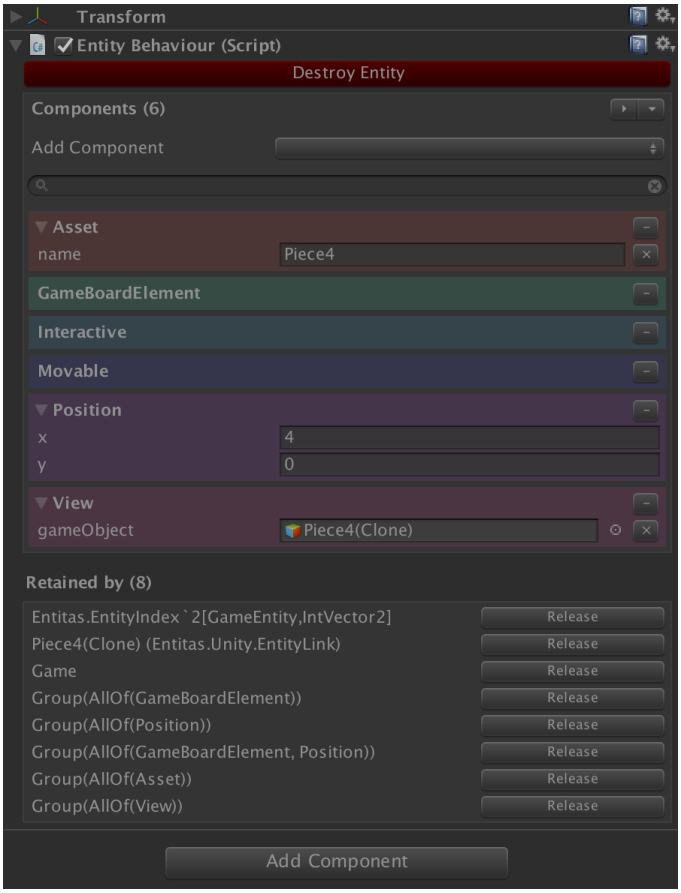
Kattavia viitekehyspaketteja, jotka hyödyntävät opinnäytetyössä esiteltyjä tapoja ohjelmoida ja jotka ovat ilmaisia sekä vapaita käyttää löytyy internetistä jaettuna useita. Joidenkin viitekehyksien kehitysten takana on yrityksiä, mutta suuri osa niistä on yksityisten henkilöiden tekemiä kokeiluja parantaa Unityllä ohjelmointia. Kun valitaan toisen kehittäjän tekemä viitekehys projektikäyttöön onkin tärkeää selvittää viitekehyksen taustat. Ensiarvoisen tärkeää on selvittää, tarjoaako viitekehyksen kehittäjä minkäänlaista tukea viitekehyksen käytölle ja onko apua saatavilla ongelmien, kuten ohjelmointivirheiden, tullessa vastaan. Alla on listattu määrittelemättömässä järjestyksessä eri viitekehyksiä Unitylle.

- Entitas https://github.com/sschmid/Entitas-CSharp
- EgoCS https://github.com/andoowhy/EgoCS/wiki
- StrangeIoC http://strangeioc.github.io/strangeioc/exec.html
- SveltoECS http://www.sebaslab.com/ecs-1-0/
- Zenobit ECS https://willhart.io/zenobits-unity-ecs-part-1/
- RobotArms https://bitbucket.org/dkoontz/robotarms
- Slash Framework https://github.com/SlashGames/slash-framework
- UniRx https://github.com/neuecc/UniRx
- uFrame ECS https://github.com/uFrame/ECS

Entitas on edellä mainituista vaihtoehtoista yksi parhaimmista viitekehyksistä sen kattavuuden ja kehittäjän antaman tuen vuoksi. Entitaksen kehityksen taustalla on saksalainen mobiilipelejä kehittävä ja julkaiseva Wooga. Viitekehyksen kehittämisen aloitti Woogalla työskentelevä insinööri Simon Schmid vuonna 2014 - aluksi firman sisäiseen käyttöön. Jo seuraavana vuonna se kuitenkin annettiin kolmansille osapuolille ilmaiskäyttöön ja samalla sen lähdekoodi tehtiin vapaaksi. Entitas on kehittäjiensä mukaan nopea, kevyt ja poistaa turhaa kompleksisuutta Unitystä. Entitas onkin kehitetty eritoten Unity-kehitystä ja C#:lla ohjelmointia varten, mutta siitä on tehty omat versionsa myös useille muille kielille, kuten C++, Java ja Haskel. (Schmid 2016)

Syitä valita Entitas ylitse muiden viitekehysten on esimerkiksi dokumentaation laatu, sillä Wooga on tehnyt muita laadukkaamman dokumentaation kehittämälleen viitekehykselle. Entitakselle on luotu oma Wiki-sivunsa sen ominaisuuksien kuvaamiseen. Tämän lisäksi Wikissä on kuvattu ohjeita perusasioiden toteuttamiseen. Entitaksella on luotu projekteja niin Woogan kuin kolmannen osapuolen toimesta ja näitä projekteja on myös esitelty Wikissä. Wooga on suurempi yhtiö, kuin muiden viitekehyksien takana olevat tahot, mikä tarjoaa käyttäjilleen laadukkaampaa ja nopeampaa palvelutukea. (Schmid 2016)

Kuten opinnäytetyössä esitellyssä harjoitustyössä, myös Entitaksessa logiikka ja tietorakenteet on eroteltu omiksi kokonaisuuksikseen. Myös entiteettikäsite on työn mukainen ja sen toiminta on lähinnä toimia säiliönä komponenteille, sekä Unity-editorissa piirtää ne samankaltaisesti, kuin esimerkkityö. Tämän lisäksi editorissa on mahdollistettu entiteetin poisto, yksittäisten komponenttien lisäys ja poisto, sekä tieto ryhmistä joissa entiteetti on. Entiteetti on myös mahdollista poistaa ryhmistä, joissa se on osallisena. Kuvassa 67 on esimerkki entiteetin tarkastelusta Unityn editorissa. (Schmid 2016)




Kuva 67. Entitaksen versio entiteetistä Unity-editorissa

Entitas hoitaa logiikan, Entitaksen tapauksessa systeemien, ja tietorakenteiden välisen keskustelun opinnäytetyön harjoitustyöstä poikkeavalla tavalla. Sen sijaan että systeemit käsittelisivät komponenttilistoja, mihin komponentit on tallennettu, pyrkii Entitas tietyllä tavalla säilyttämään olio-ohjelmoinnin mukaisen olio-lähtöisen lähestymistavan. Systeemien ja komponenttien välille on luotu ylimääräinen taso ryhmät ja systeemit käsittelevät näitä ryhmiä, oletuksena yksi systeemi yhtä ryh-

mää varten. Ryhmät koostuvat entiteeteistä, joita muokataan ja entiteetin kuuluminen ryhmään määräytyy sen mukaan, mitä komponentteja se pitää sisällään. (Schmid S. 2016)

Entitaksen systeemit on jaettu kolmeen erilliseen luokkaan, jotka ovat IInitializeSystem, IExecuteSystem ja IReactiveSystem. Nämä luokat eroavat toisistaan niiden ajotavalla. Systeemiluokista IInitializeSystem, ajetaan läpi vain kerran ensimmäisellä kehyksellä, jolloin se on luotu. IExecuteSystem taas vastaavasti ajetaan jokaisessa kehyksessä. IReactiveSystem poikkeaa näistä kahdesta toiminnaltaan, eikä se ole kehyksistä riippuvainen. Sen sijaan se ajetaan aina, kun ryhmä jonka muokkaamiseen tällainen luokka on tarkoitettu, muuttuu jollain tavalla. (Schmid S. 2016)

Entitas tarjoaa edellä esiteltyjen luokkien ja metodien lisäksi myös muita työkaluja. Siitä löytyy valmiiksi muun muassa varantotyökalu, jonka avulla entiteettejä on helppo luoda ja tuhota ajon aikana. Entitas tarjoaa Unity-käyttäjilleen myös koodigeneroijan, joka osaa luoda nappia painamalla erilaisia luokkia ja funktioita valmiiksi. Entitaksessa on myös beta-tasolla oleva ominaisuus, jolla voidaan luoda entiteeteistä Blueprint-objekteja. Entitaksessa Blueprint-objekti on tapa luoda valmis entiteetti komponentteineen ja oikeine arvoineen vain kirjoittamalla BinaryFormatter-tiedosto ja raahaamalla se Unity-editoriin. Käyttäjä voi halutessaan käyttää ominaisuuden hyödyntämiseen myös muita tiedostoformaatteja, kuten JSON ja Xml. (Schmid 2016)

Entitasta, tai jotain muuta viitekehystä käyttämällä saadaan nopeasti parannettua Unityn arkkitehtuuria. Tämä helpottaa varsinkin suurempien projektien luomista ja ylläpitoa, joka pitemmällä tähtäimellä säästää kehitysresursseja. Valmiit viitekehykset kuitenkin vaativat toisen tekemän ohjelmiston opettelun, joka voi laajan viitekehyksen tapauksessa olla pitkäkin aika. Valmiin viitekehyksen käyttö myös siirtää osan koodin vastuusta viitekehyksen luojalle, ja jos viitekehyksestä on ohjelmointivirheitä, voi loppukäyttäjän olla haastava niitä löytää. Viitekehyksen ohjelmointivirheiden tapauksessa käyttäjän on joskus pyydettävä viitekehyksen tekijää korjaamaan ongelmat, mikä vaatii sitä enemmän aikaa, mitä huonomman tuen vii-

tekehyksen tekijä tarjoaa tuotteelleen. Valmista viitekehystä käyttämällä loppu-käyttäjä ei myöskään voi muokata Unityn toimintatapoja juuri sellaiseksi kuin itse haluaa niin hyvin kuin oman viitekehyksen luomalla.

# 6 YHTEENVETO

Opinnäytetyön tavoitteena oli tutkia parempi tapoja ohjelmoida Unityllä. Unityn arkkitehtuuri pohjautuu olio-ohjelmoinnin periaatteisiin, joka ei ole nykyaikana tehokkain tapa ohjelmoida niin suorituskyvyn kuin projektinhallinnankaan kannalta. Tämä vuoksi päätettiin tutkia vaihtoehtoisia ohjelmointimalleja, joiksi valikoitui dataorientoitunut suunnittelu ja funktionaalinen ohjelmointi. Nämä mallit valittiin, koska kumpaakin mallia hyödyntävät kehitystyökalut ovat yleistyneet viime vuosina ja niiden eduista on viime aikoina virinnyt keskustelua kehittäjien kesken.

Teoriaosuudessa tarkastelu aloitettiin pelienkehityksen nykytilanteesta: Minkälaisia nykyteknologian erityistarpeet ovat, sekä millä tavalla olio-ohjelmointi, ja sen periaatteisiin pohjautuva Unity-kehitysympäristö, vastaavat pelinkehityksen haasteisiin. Työssä havaittiin, että olio-ohjelmoinnille ja muille Unityssä käytettäville ohjelmointimalleille on olemassa parempia vaihtoehtoja. Niitä käyttämällä voidaan saada aikaan modulaarisuutta ja selkeyttä, jota olio-ohjelmoinnin on vaikea toteuttaa.

Dataorientoituneen suunnittelun tietoperustaksi perustui pelinkehittäjien blogikirjoitukset, artikkelit ja esitykset, koska perinteistä kirjallisuutta aiheesta ei vielä ole olemassa. Päälähteenä dataorientoitunut suunnittelu -kappaleeseen valikoitui Richard Fabianin ”Data-oriented Design” -e-kirja, koska se oli laajin kirjoitettu kokonaisuus aiheesta – sekä siihen oli viitattu useissa aiheeseen liittyvissä artikkeleissa. Funktionaalisen ohjelmoinnin kokonaisuus kerättiin käyttämällä C#:iin ja muihin kieliin perustuvia kirjoja ja tiedeartikkeleita. Funktionaalinen ohjelmointikappaleen päälähteenä olivat Oliver Sturmin kirjoittama ”Functional Programming in C#: Classic Programming Techniques for Modern Projects” ja Tomas Petricekin kirjoittama ”Real-World Functional Programming With examples in F# and C#”, koska ne olivat Amazon- ja BookDepository-verkkokauppojen myydyimpien C#:iin liittyvien funktionaalisen ohjelmoinnin kirjojen joukossa.

Havaittiin, että molemmat mallit, dataorientoitunut suunnittelu ja funktionaalinen ohjelmointi, eroavat hyvin paljon olio-ohjelmoinnista. Tämän vuoksi mallien oppiminen ja sisäistäminen voi olla vaikeaa. Ohjelmointimallien oppimista tukemaan

kirjoitettiin koodiesimerkkejä, joita on sekä kolmannessa että neljännessä kappaleessa. Vaikean opittavuuden lisäksi molemmissa malleissa huomattiin myös muita vähäpätöisempiä ongelmia, mutta samalla niillä havaittiin olevan niin suuria etuja olio-ohjelmointiin nähden, että niiden käyttäminen toisi selkeitä etuja peliohjelmoijalle. Opinnäytetyöprosessin aikana selvisi, että ohjelmointimallit tuovat modulaarisuutta ja hallinnoitavuutta koodiin. Lisäksi, mallit helpottavat rinnakkaisohjelmointia. Edellä mainittujen lisäksi dataorientoituneen suunnittelun havaittiin tehostavan muistinkäyttöä.

Tietoperustaa hyödynnettiin käytännön osuudessa uuden viitekehyksen luomiseen Unityssä C#:lla. Viitekehystä toteutettaessa huomioitiin myös Unityn vahvuudet. Dataorientoituneen suunnittelun malleja käytettiin tietorakenteiden mallintamiseen ja funktionaalista ohjelmointia tietorakenteiden hallintaan, eli pelilogiikkaan. Viitekehystä verrattiin kahteen muuhun malliin käyttämällä Unityn Profiler-työkalua, ja siitä saatavia arvoja.

Dataorientoituneeseen suunnitteluun ja funktionaaliseen ohjelmointiin perustuvan viitekehyksen havaittiin olevan Unityn MonoBehaviour-toteutusta huomattavasti tehokkaampi, sekä helppokäyttöisempi. Profiler-työkalulla testattiin objektien liikkumisen laskentaan, luomiseen ja tuhoamiseen käytettyä aikaa. Huomattiin, että viitekehyksen tekeminen on vaativaa, mutta valmis viitekehys helpottaa huomattavasti ohjelmoijan työtä, ja siten nopeuttaa pelinkehitystä. Tärkeää on löytää sopiva tasapaino joustavuuden ja tehokkuuden välillä.

Tutkimustulosten voidaan katsoa olevan luotettavia, ainakin esimerkeissä käytettyjen ohjelmointimallien keskinäisessä suhteessa. Ohjelmointimallien tietorakenteet ja toimintalogiikka pyrittiin ohjelmoimaan mahdollisimman samankaltaisesti toimiviksi, jotta ylimääräisiä viittausten hakuja tai muita mahdollisia poikkeavuuksien aiheuttajia ei ilmenisi. Objektien liikkumiseen käytetystä laskennasta otettiin tarpeeksi pitkä testijakso, jolloin saatiin minimoitua testistä ulkoiset tekijät sekä mahdolliset anomaliat. Laskettujen keskiarvojen voi olettaa antavan luotettavan kuvan ohjelmointimallien välisistä tehokkuuseroista.

Opinnäytetyötä tehdessä heräsi useita jatkokehitys- ja tutkimusideoita. Vaikka Unity ei suoraan tuekaan F#-ohjelmointikielellä kehittämistä, pystyy sitä käyttämään Unityssä siten, että F#-projektista luodaan dll-tiedosto, josta Unity tunnistaa funktiot. F# on funktionaalinen kieli, joten se soveltuu erinomaisesti esimerkiksi tässä opinnäytetyössä esitellyn Fudo-mallin prosessoreiden toteuttamiseen.

Pidemmälle tietorakenteiden hallinnassa voisi mennä tutkimalla aihetta ”Data-driven Design”, joka on samantyylinen dataorientoituneen suunnittelun kanssa, mutta jossa kaikki rakentuu tietorakenteisiin keskittyneesti. Data-driven-oppien mukaisesti entiteetit voitaisiin tallentaa JSON-tiedostoon, josta lukemalla luodaan peliobjektit Unityn Prefab-mallin mukaisesti.

Myös entiteettien toteuttamista Unityssä voisi kehittää pidemmälle. Esimerkiksi Unityn käyttöliittymään voisi rakentaa lisäosan, jossa tietorakennekomponentteja voisi luoda ja poistaa napin painalluksella. Lisäksi entiteettien piirtämistapaa voisi laajentaa esimerkiksi väreillä. Entiteettien hallintaa voisi automatisoida niin, että entiteetti tarkistaa siihen linkitetyt komponentit saman tyyppistä listoista, ja jos komponenttia ei ole, entiteetti poistaa automaattisesti komponenttityyppilistastaan kyseisen komponenttityypin. Tämä poistaa tarpeen piirtää komponentti editorinäkymässä, joka parantaa suorituskykyä.

Tapahtumahallintaluokan toimintaa voisi laajentaa ottamalla huomioon esimerkiksi enimmäismäärän samalla kehyksellä ajettavista tapahtumista. Tapahtumat voitaisiin luokitella myös tärkeysjärjestykseen helpottamaan niiden hallintaa. Käyttöliittymässä tapahtuvat toiminnot voitaisiin järjestää omaan tapahtumahallintaluokkaansa. Eli, kun käyttäjä antaa käyttöliittymässä syötteen, se laukaistaan tapahtumana jossain pelilogiikassa, jolloin ylimääräisiä viittauksia ei tarvita ja käyttäjän syötteen vastaanottaminen yksinkertaistuu.

Prosessoreiden toimintaa voisi myös laajentaa ottamalla huomioon erilaisia rajatapaustilanteita. Esimerkiksi tapaukset joissa tieto ei muutu kehysten välillä voitaisiin jättää käymättä läpi. Esimerkkinä tästä on tapaus jossa objektin nopeus ja suunta ei muutu, joten objektin ei sitä tarvitse laskea vauhtia joka kehys uudelleen.

Opinnäytetyössä esitettyjä ohjelmointimalleja voi soveltaa missä tahansa peliprojektissa pelimoottorista riippumatta. Tehtyä viitekehystä voi käyttää myös Unityn

ulkopuolella, kunhan esitellystä mallista korvataan Unityn omat tietorakenteet kohteen omilla tai itsetehdyillä tietorakennetyypeillä.

Opinnäytetyön tavoitteet täyttyivät. Uusista ohjelmointimalleista saatiin selville hyödyt ja haitat sekä niitä käytettiin onnistuneesti oman viitekehyksen ohjelmoimiseen. Dataorientoituneen suunnittelun ja funktionaalisen ohjelmoinnin yhteensopivuutta Unity-kehitysympäristöön ei ole dokumentoitu tässä laajuudessa aiemmin, joten opinnäytetyöprosessi kehitti huomattavasti omaa osaamista aiheesta. Tämä opinnäytetyö tarjoaa myös muille hyvän pohjan perehtyä aiheeseen ja lisätä tietoutta vaihtoehtoisista tavoista ohjelmoida Unityllä. Opinnäytetyön aikana toteutettu viitekehys on saatavilla osoitteesta: https://github.com/t4r4ntuli/fudo.

LÄHTEET

Akhmechet, Slava. (19.6.2006). Functional Programming for The Rest of Us. Saatavilla: http://www.defmacro.org/ramblings/fp.html#part_4 (Lisätty 27.3.2017)

Nimimerkki Atehwa. (2013). Ohjelmointiparadigmat. Saatavilla: http://sange.fi/~atehwa/cgi-bin/piki.cgi/ohjelmointiparadigmat (Lisätty: 22.2.2017)

Bennet S, McRobb S. & Farmer R. (2006). Object-Oriented Systems Analysis and Design using UML. Coventry University. McGraw-Hill. (Lisätty: 21.2.2017)

Boreal Games. (2.4.2013). Understanding Component-Entity-Systems. Saatavilla: https://www.gamedev.net/resources/_/technical/game-programming/understanding-component-entity-systems-r3013 (Lisätty: 10.3.2017)

Brumme, Chris. (2004). Why doesn't C# support multiple inheritance? Saatavilla: https://blogs.msdn.microsoft.com/csharpfaq/2004/03/07/why-doesnt-c-support-multiple-inheritance/ (Lisätty 16.4.2017)

Carmack, John. (30.4.2012). In-depth: Functional programming in C++. Saatavilla: http://www.gamasutra.com/view/news/169296/Indepth_Functional_programming_in_C.php (Lisätty: 20.3.2017)

Clark, Dan. (2011). Beginning C# Object-Oriented Programming. Apress.

Cook, Mary Rose. (2013). A Practical Introduction to Functional Programming. Saatavilla: https://maryrosecook.com/blog/post/a-practical-introduction-to-functional-programming (Lisätty 27.3.2017)

DICE. (2010). Introduction to Data-Oriented Design. Saatavilla: http://www.dice.se/wp-content/uploads/2014/12/Introduction_to_Data-Oriented_Design.pdf (Lisätty: 20.2.2017)

Fabian, Richard. (25.6.2013). Data-oriented Design. Saatavilla: http://www.data-orienteddesign.com/dodmain/node3.html (Lisätty: 15.2.2017)

Harrop, Jon. (30.5.2016). Disadvantages of purely functional programming. Saatavilla: http://flyingfrogblog.blogspot.fi/2016/05/disadvantages-of-purely-functional.html (Lisätty: 29.3.2017)

Jelvis, Tikhon. (1.4.2014). What are some limitations/disadvantages of functional programming? Saatavilla: https://www.quora.com/What-are-some-limitations-disadvantages-of-functional-programming (Lisätty 29.3.2017)

Kotaku. (30.11.2016). Nearly 40% of all Steam Games Were Released in 2016. Saatavilla: http://kotaku.com/nearly-40-of-all-steam-games-were-released-in-2016-1789535450 (Lisätty: 20.2.2017)

Leonard, Tom. (1998). Postmortem: Thief: The Dark Project. Saatavilla: http://www.gamasutra.com/view/feature/3355/post___mortem_thief_the_dark_project.php?print=1 (Lisätty: 6.3.2017)

Leveson, Nancy & Turner, Clark. (1993). An Investigation of the Therac-25 Accidents. Saatavilla: http://www.cs.umd.edu/class/spring2003/cmsc838p/Misc/therac.pdf (Lisätty 8.4.2017)

Llopis, Noel. (9.2009). Data-Oriented Design (Or Why You Might Be Shooting Yourself in the Foot with OOP). Saatavilla: http://gamesfromwithin.com/data-oriented-design (Lisätty 15.2.2017)

Llopis, Noel. (9.2010). Data-Oriented Design Now and in the Future. Saatavilla: http://gamesfromwithin.com/data-oriented-design-now-and-in-the-future (Lisätty 16.3.2017)

Mandalà, Sebastiano. (30.9.2012). Inversion of Control with Unity 3D – part 1. Saatavilla: https://www.sebaslab.com/ioc-container-for-unity3d-part-1/ (Lisätty: 6.2.2017)

Mandalà, Sebastiano. (25.4.2013). What's wrong with Unity SendMessage and BroadcastMessage and what to do about it. Saatavilla: https://www.sebaslab.com/whats-wrong-with-sendmessage-and-broadcastmessage-and-what-to-do-about-it/ (Lisätty: 11.2.2017)

Mandalà, Sebastiano. (5.10.2015). Is Unity 3D a Real Entity Component System Engine? Saatavilla: https://www.linkedin.com/pulse/inversion-control-entity-component-systems-unity-engine-mandal%C3%A0 (Lisätty: 6.2.2017)

McCarthy, John. (6.1978) History of Lisp in ACM/SIGPLAN History of Programming Languages Conference: 217-223. (Lisätty: 11.4.2017)

Microsoft Corporation (2012). Description of race conditions and deadlocks. Saatavilla: https://support.microsoft.com/en-us/help/317723/description-of-race-conditions-and-deadlocks (Lisätty 21.3.2017)

Microsoft Corporation (2015). Refactoring Into Pure Functions (C#). Saatavilla: https://msdn.microsoft.com/en-us/library/mt693174.aspx (Lisätty: 21.3.2017)

Microsoft Corporation. (2017). IComponent Interface. Saatavilla: https://msdn.microsoft.com/en-us/library/system.componentmodel.icomponent(v=vs.110).aspx?cs-save-lang=1&cs-lang=csharp#code-snippet-2 (Lisätty: 16.3.2017)

Microsoft Corporation. (2017). IEnumerable Interface. Saatavilla: https://msdn.microsoft.com/en-us/library/system.collections.ienumerable%28v=vs.110%29.aspx?f=255&MSPPError=-2147217396) (Lisätty: 23.4.2017)

Microsoft Corporation (2017). String Class. Saatavilla: https://msdn.microsoft.com/en-us/library/system.string(v=vs.110).aspx#Immutability (Lisätty 6.4.2017)

Microsoft Corporation (2017). Tuple Class. Saatavilla: https://msdn.microsoft.com/en-us/library/system.tuple(v=vs.110).aspx (Lisätty 6.4.2017)

Milewski, Bartosz. (9.4.2012). The Downfall of Imperative Programming. https://www.fpcomplete.com/blog/2012/04/the-downfall-of-imperative-programming (Lisätty: 21.3.2017)

Mitton, Richard. (17.6.2015) Explaining Data Oriented Design. Saatavilla: http://www.codersnotes.com/notes/explaining-data-oriented-design/ (Lisätty: 15.2.2017)

Nystrom, Robert. (2014). Game Programming Patterns. USA: Genever Benning.

Oeing, Christian. (16.8.2015). Why to use a custom CBES architecture within Unity. Saatavilla: http://unity-coding.slashgames.org/why-to-use-a-custom-cbes-architecture-within-unity/ (Lisätty 6.2.2017)

Petricek, Tomas. (2009). Real-World Functional Programming With examples in F# and C#. New York, USA: Manning Publications.

Popovic, Jovan. (10.6.2012). Functional programming in C#. Saatavilla: https://www.codeproject.com/Articles/375166/Functional-programming-in-Csharp (Lisätty: 27.3.2017)

Porter, Nicolas. (2012). Component-based game object system. Saatavilla: https://raw.githubusercontent.com/surjikal/cbgos-experiment/master/doc/nico-lasportercbgos-paper.pdf (Lisätty: 20.2.2017)

ProgrammerInterview. (2016). In C++, what's the diamond problem, and how can it be avoided? Saatavilla: http://www.programmerinterview.com/index.php/c-cplusplus/diamond-problem/ (Lisätty: 25.2.2017)

Rezaei, Pedram. (2.6.2017). Lazy evaluation in C# Saatavilla: https://blogs.msdn.microsoft.com/pedram/2007/06/02/lazy-evaluation-in-c/ (Lisätty 6.4.2017)

Sefton, Daniel. (7.5.2016). Developing a Data-Oriented Game Engine. Saatavilla: http://www.danielsefton.com/2016/05/developing-a-data-oriented-game-engine-part-1/ (Lisätty: 20.3.2017)

Sergey Galyonkin. (30.11.2016). Steam Spy: 38% of all Steam games were released in 2016. https://twitter.com/Steam_Spy/status/804072335997358084/photo/1?ref_src=twsrc%5Etfw (Lisätty: 10.4.2017)

Schmid, Simon. (8.6.2017): Entitas - The Entity Component System Framework for C# and Unity. Saatavilla: https://github.com/sschmid/Entitas-CSharp (Lisätty 7.5.2017)

Simonov, Valentin. (23.12.2015). 10000 Update() Calls. Saatavilla: https://blogs.unity3d.com/2015/12/23/1k-update-calls/ (Lisätty: 11.2.2017)

Sorva, Juha. (2016). Olio-ohjelmointi. Saatavilla: https://plus.cs.hut.fi/o1/2016/k02/osa01/ (Lisätty: 20.2.2017)

Sturm, Oliver. (2011). Functional Programming in C#: Classic Programming Techniques for Modern Projects. https://books.google.fi/books?id=nOc-FYPXN9IC&printsec=frontcover&hl=fi#v=onepage&q&f=false

Suleman, Aater. (20.5.2011). What makes parallel programming hard? http://www.futurechips.org/tips-for-power-coders/parallel-programming.html (Lisätt: 8.4.2017)

Sweeney, Mark. (7.10.2014). Unity3D: Direct Funtion calling vs SendMessage. Saatavilla: http://www.mark-sweeney.com/?p=822 (Lisätty: 11.2.2017)

The Saylor Foundation. (2010). Advantages and Disadvantages of Object-Oriented Programming. Saatavilla: https://www.saylor.org/site/wp-content/uploads/2013/02/CS101-2.1.2-AdvantagesDisadvantagesOfOOP-FINAL.pdf (Lisätty: 19.2.2017)

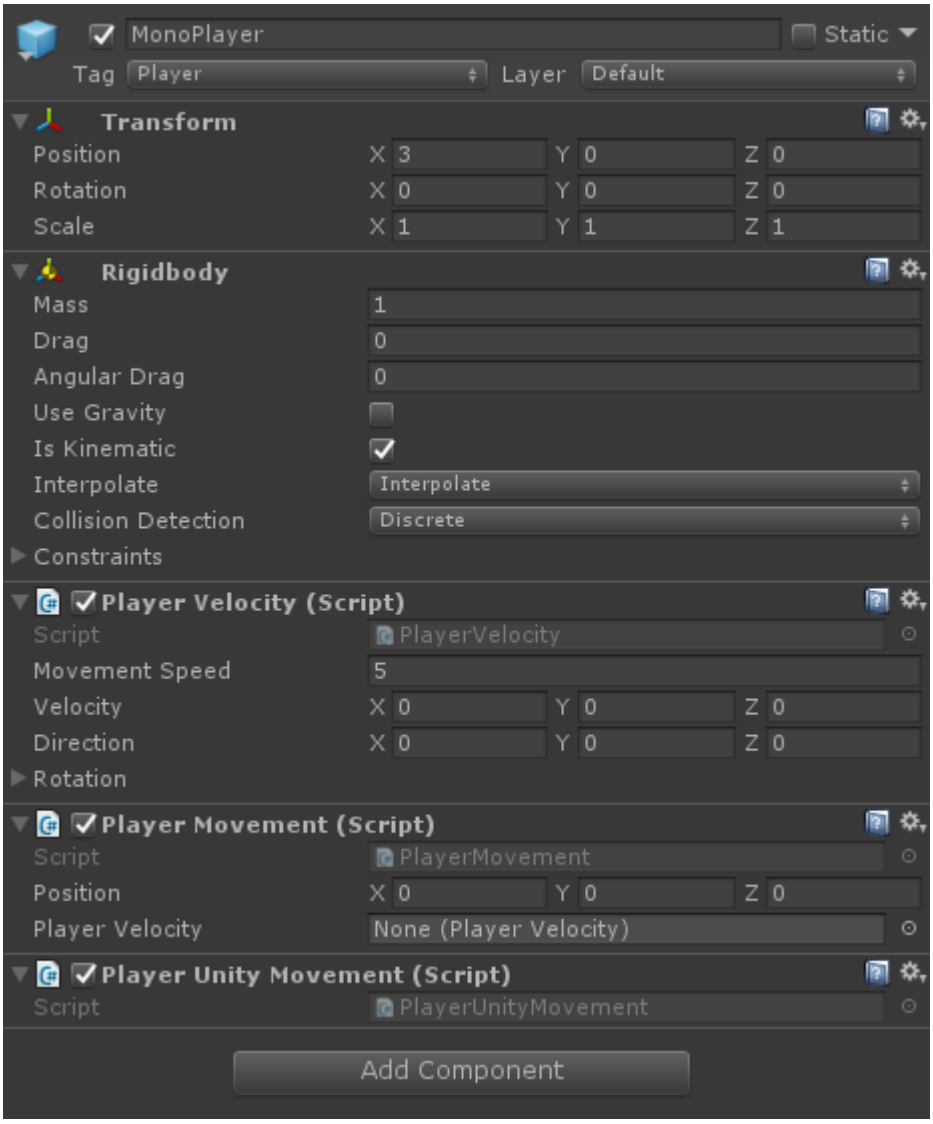
Totin, Alexey. (2016). Memory Problems in .NET. JetBrains.

Unity Technologies. (2017). Unity – Fast Facts. Saatavilla: https://unity3d.com/public-relations (Lisätty 11.2.2017)

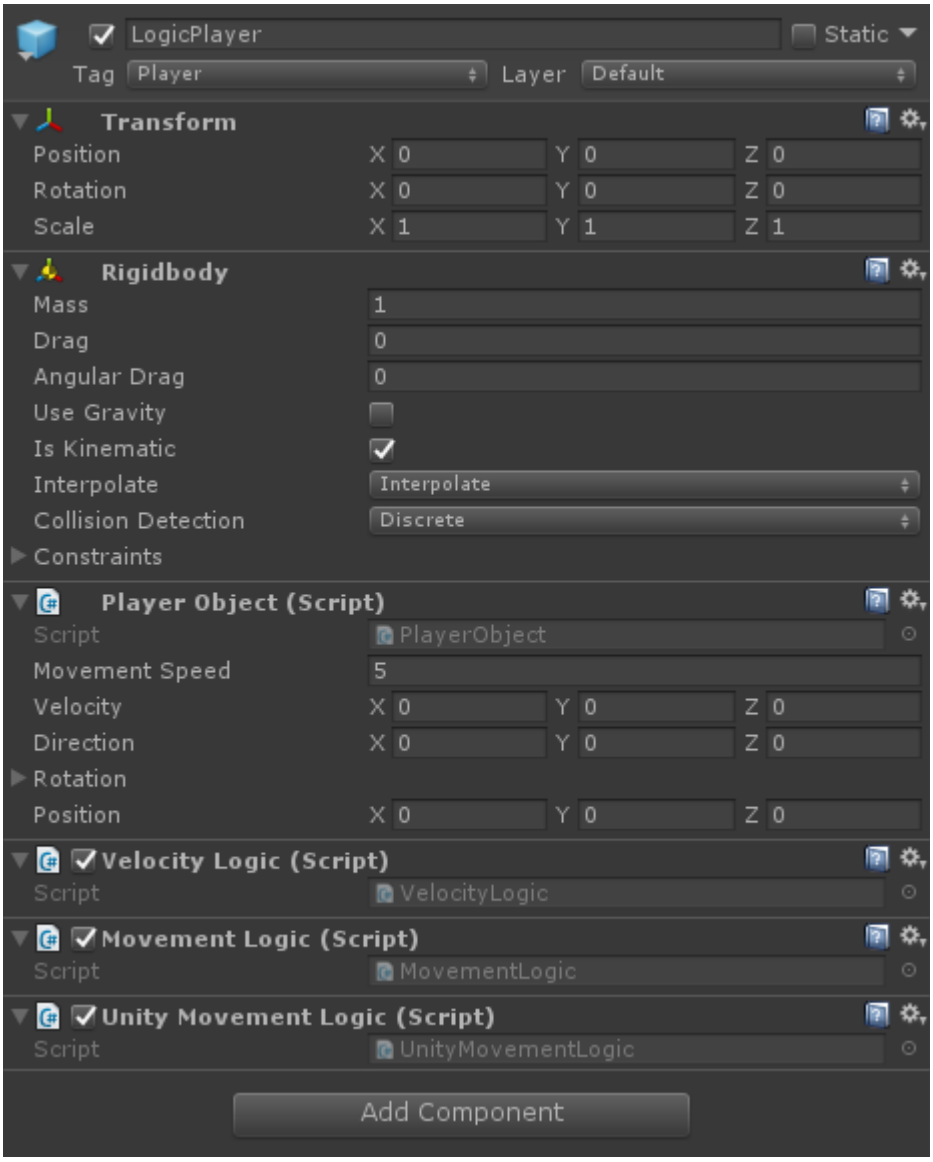
Unity Technologies. (2017). Coroutines. Saatavilla: https://docs.unity3d.com/Manual/Coroutines.html (Lisätty: 23.4.2017)

Unity Technologies. (2017). Unity. Saatavilla: https://unity3d.com/unity (Lisätty 11.2.2017)

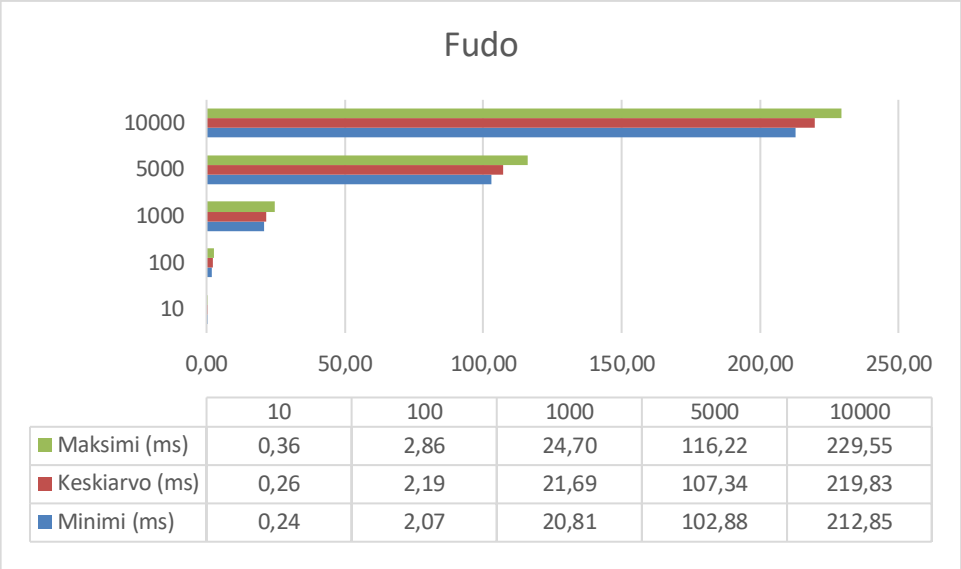
Liite 1. MonoBehaviour-luokan päivitysfunktioita käyttävä pelaajaobjekti



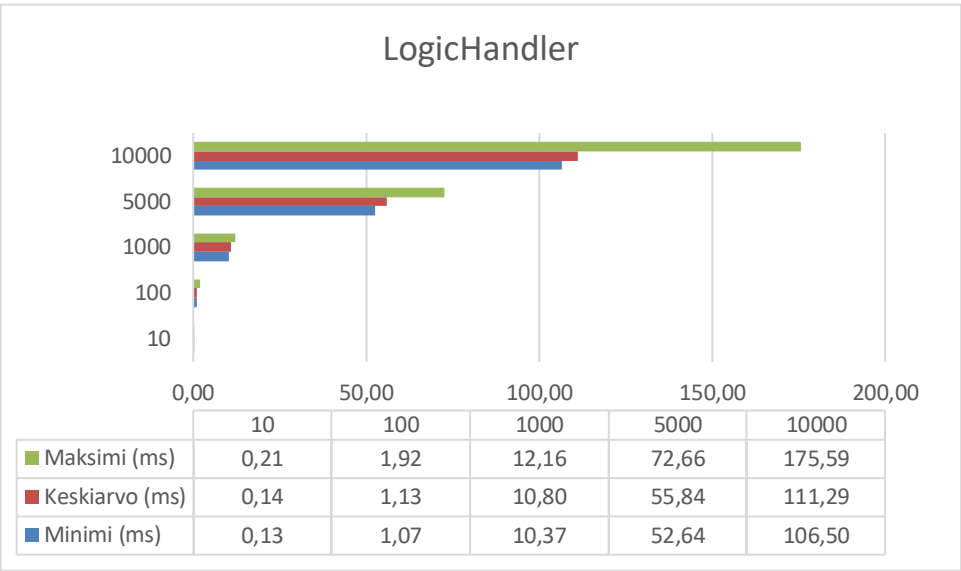
Liite 2. LogicHandler-luokan päivitysfunktioita käyttävä pelaajaobjekti



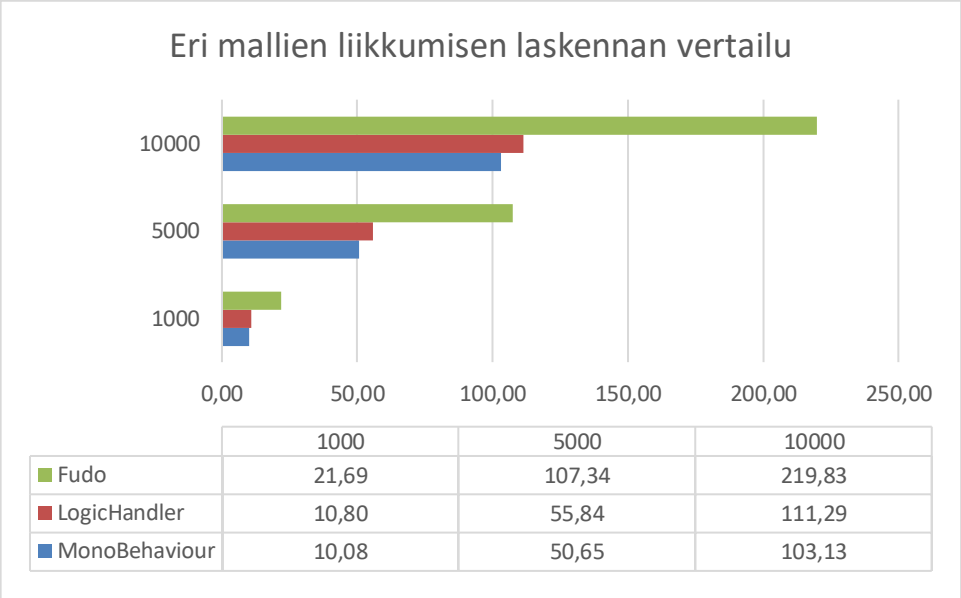
Liite 3. Fudo- ja LogicHandler- mallien testitulokset sekä testitulosten vertailu Profiler-työkalun Deep Profile -asetuksen ollessa päällä.




| | 10 | 100 | 1000 | 5000 | 10000 |
|---|---|---|---|---|---|
| Maksimi (ms) | 0,36 | 2,86 | 24,70 | 116,22 | 229,55 |
| Keskiarvo (ms) | 0,26 | 2,19 | 21,69 | 107,34 | 219,83 |
| Minimi (ms) | 0,24 | 2,07 | 20,81 | 102,88 | 212,85 |




| | 10 | 100 | 1000 | 5000 | 10000 |
|---|---|---|---|---|---|
| Maksimi (ms) | 0,21 | 1,92 | 12,16 | 72,66 | 175,59 |
| Keskiarvo (ms) | 0,14 | 1,13 | 10,80 | 55,84 | 111,29 |
| Minimi (ms) | 0,13 | 1,07 | 10,37 | 52,64 | 106,50 |




| | 1000 | 5000 | 10000 |
|---|---|---|---|
| Fudo | 21,69 | 107,34 | 219,83 |
| LogicHandler | 10,80 | 55,84 | 111,29 |
| MonoBehaviour | 10,08 | 50,65 | 103,13 |

# Liite 4. Ajon aikaisten suorituskykymittausten tulokset Deep Profile päällä

Laskenta-aika per kehys

| MonoBehaviour | | | | | |
|---|---|---|---|---|---|
| Count | 10 | 100 | 1000 | 5000 | 10000 |
| Minimi (ms) | 0,07 | 0,96 | 9,48 | 48,58 | 98,49 |
| Keskiarvo (ms) | 0,13 | 1,04 | 10,08 | 50,65 | 103,13 |
| Maksimi (ms) | 0,24 | 1,33 | 11,52 | 59,38 | 110,05 |

| LogicHandler | | | | | |
|---|---|---|---|---|---|
| Lukumäärä | 10 | 100 | 1000 | 5000 | 10000 |
| Minimi (ms) | 0,13 | 1,07 | 10,37 | 52,64 | 106,50 |
| Keskiarvo (ms) | 0,14 | 1,13 | 10,80 | 55,84 | 111,29 |
| Maksimi (ms) | 0,21 | 1,92 | 12,16 | 72,66 | 175,59 |

| Fudo | | | | | |
|---|---|---|---|---|---|
| Lukumäärä | 10 | 100 | 1000 | 5000 | 10000 |
| Minimi (ms) | 0,24 | 2,07 | 20,81 | 102,88 | 212,85 |
| Keskiarvo (ms) | 0,26 | 2,19 | 21,69 | 107,34 | 219,83 |
| Maksimi (ms) | 0,36 | 2,86 | 24,70 | 116,22 | 229,55 |

Jätekuorman laskenta-aika

| MonoBehaviour | | | | | |
|---|---|---|---|---|---|
| Lukumäärä | 10 | 100 | 1000 | 5000 | 10000 |
| Minimi (ms) | 0,06 | 0,07 | 0,09 | 0,19 | 0,35 |
| Keskiarvo (ms) | 0,09 | 0,10 | 0,14 | 0,25 | 0,45 |
| Maksimi (ms) | 0,15 | 0,17 | 0,24 | 0,36 | 0,80 |

| LogicHandler | | | | | |
|---|---|---|---|---|---|
| Lukumäärä | 10 | 100 | 1000 | 5000 | 10000 |
| Minimi (ms) | 0,06 | 0,06 | 0,09 | 0,18 | 0,31 |
| Keskiarvo (ms) | 0,09 | 0,10 | 0,14 | 0,26 | 0,42 |
| Maksimi (ms) | 0,16 | 0,20 | 0,23 | 0,44 | 0,90 |

| Fudo | | | | | |
|---|---|---|---|---|---|
| Lukumäärä | 10 | 100 | 1000 | 5000 | 10000 |
| Minimi (ms) | 0,06 | 0,06 | 0,09 | 0,19 | 0,31 |
| Keskiarvo (ms) | 0,08 | 0,10 | 0,15 | 0,25 | 0,39 |
| Maksimi (ms) | 0,16 | 0,26 | 0,24 | 0,40 | 0,75 |

Liite 5. Ajon aikaisten suorituskykymittausten tulokset Deep Profile pois päältä

Laskenta-aika per kehys

| MonoBehaviour | | | | | |
|---|---|---|---|---|---|
| Lukumäärä | 10 | 100 | 1000 | 5000 | 10000 |
| Minimi (ms) | 0,07 | 0,96 | 9,48 | 48,58 | 98,49 |
| Keskiarvo (ms) | 0,13 | 1,04 | 10,08 | 50,65 | 103,13 |
| Maksimi (ms) | 0,24 | 1,33 | 11,52 | 59,38 | 110,05 |

| LogicHandler | | | | | |
|---|---|---|---|---|---|
| Lukumäärä | 10 | 100 | 1000 | 5000 | 10000 |
| Minimi (ms) | 0,01 | 0,06 | 0,57 | 3,23 | 6,57 |
| Keskiarvo (ms) | 0,01 | 0,07 | 0,67 | 3,57 | 7,08 |
| Maksimi (ms) | 0,04 | 0,15 | 1,53 | 4,50 | 8,82 |

| Fudo | | | | | |
|---|---|---|---|---|---|
| Lukumäärä | 10 | 100 | 1000 | 5000 | 10000 |
| Minimi (ms) | 0,01 | 0,08 | 0,86 | 4,38 | 9,19 |
| Keskiarvo (ms) | 0,02 | 0,11 | 0,95 | 4,64 | 9,87 |
| Maksimi (ms) | 0,05 | 0,32 | 1,20 | 5,24 | 13,05 |

Jätekuorman laskenta-aika

| MonoBehaviour | | | | | |
|---|---|---|---|---|---|
| Lukumäärä | 10 | 100 | 1000 | 5000 | 10000 |
| Minimi (ms) | 0,06 | 0,07 | 0,09 | 0,19 | 0,35 |
| Keskiarvo (ms) | 0,09 | 0,10 | 0,14 | 0,25 | 0,45 |
| Maksimi (ms) | 0,15 | 0,17 | 0,24 | 0,36 | 0,80 |

| LogicHandler | | | | | |
|---|---|---|---|---|---|
| Lukumäärä | 10 | 100 | 1000 | 5000 | 10000 |
| Minimi (ms) | 0,05 | 0,05 | 0,00 | 0,10 | 0,15 |
| Keskiarvo (ms) | 0,09 | 0,08 | 0,06 | 0,15 | 0,20 |
| Maksimi (ms) | 0,25 | 0,17 | 0,29 | 0,39 | 0,56 |

| Fudo | | | | | |
|---|---|---|---|---|---|
| Lukumäärä | 10 | 100 | 1000 | 5000 | 10000 |
| Minimi (ms) | 0,05 | 0,05 | 0,05 | 0,09 | 0,12 |
| Keskiarvo (ms) | 0,09 | 0,09 | 0,08 | 0,13 | 0,18 |
| Maksimi (ms) | 0,26 | 0,23 | 0,22 | 0,40 | 0,33 |

# Liite 6. Objektien luomisen suorituskykymittausten tulokset

Ilman ajon aikaista AddComponent-funktion käyttöä

| MonoBehaviour | | | | | | |
|---|---|---|---|---|---|---|
| Lukumäärä: 1000 | Otos 1 | Otos 2 | Otos 3 | Otos 4 | Otos 5 | Keskiarvo |
| Aika (ms) | 42,07 | 43,23 | 32,89 | 33,47 | 33,00 | 36,93 |
| Jätekuorma (kb) | 281,4 | 281,4 | 281,4 | 281,4 | 281,4 | 281,40 |

| LogicHandler | | | | | | |
|---|---|---|---|---|---|---|
| Lukumäärä: 1000 | Otos 1 | Otos 2 | Otos 3 | Otos 4 | Otos 5 | Keskiarvo |
| Aika (ms) | 40,99 | 41,29 | 40,49 | 40,51 | 41,61 | 40,98 |
| Jätekuorma (kb) | 420,80 | 420,80 | 420,80 | 420,80 | 420,80 | 420,80 |

| Fudo | | | | | | |
|---|---|---|---|---|---|---|
| Lukumäärä: 1000 | Otos 1 | Otos 2 | Otos 3 | Otos 4 | Otos 5 | Keskiarvo |
| Aika (ms) | 35,16 | 34,52 | 40,78 | 46,66 | 34,79 | 38,38 |
| Jätekuorma (kb) | 900,00 | 900,00 | 900,00 | 900,00 | 900,00 | 900,00 |

Ajon aikaisen AddComponent-funktion käytön kanssa

| MonoBehaviour | | | | | | |
|---|---|---|---|---|---|---|
| Lukumäärä: 1000 | Otos 1 | Otos 2 | Otos 3 | Otos 4 | Otos 5 | Keskiarvo |
| Aika (ms) | 142,31 | 146,18 | 130,04 | 140,81 | 140,81 | 140,03 |
| Jätekuorma (kb) | 4740,00 | 4740,00 | 4740,00 | 4740,00 | 4740,00 | 4740,00 |

| LogicHandler | | | | | | |
|---|---|---|---|---|---|---|
| Lukumäärä: 1000 | Otos 1 | Otos 2 | Otos 3 | Otos 4 | Otos 5 | Keskiarvo |
| Aika (ms) | 181,57 | 176,06 | 179,04 | 175,68 | 174,15 | 177,30 |
| Jätekuorma (kb) | 8816,00 | 8816,00 | 8816,00 | 8816,00 | 8816,00 | 8816,00 |

Liite 7. Objektien tuhoamisen suorituskykymittausten tulokset

| MonoBehaviour | | | | | | |
|---|---|---|---|---|---|---|
| Lukumäärä: 1000 | Otos 1 | Otos 2 | Otos 3 | Otos 4 | Otos 5 | Keskiarvo |
| Aika (ms) | 15,13 | 19,85 | 14,92 | 15,33 | 14,91 | 16,03 |
| Jätekuorma (kb) | 7,9 | 7,9 | 7,9 | 7,9 | 7,9 | 7,90 |

| LogicHandler | | | | | | |
|---|---|---|---|---|---|---|
| Lukumäärä: 1000 | Otos 1 | Otos 2 | Otos 3 | Otos 4 | Otos 5 | Keskiarvo |
| Aika (ms) | 23,35 | 23,37 | 23,60 | 23,75 | 22,97 | 23,41 |
| Jätekuorma (kb) | 8 | 8 | 8 | 8 | 8 | 8,00 |

| Fudo | | | | | | |
|---|---|---|---|---|---|---|
| Lukumäärä: 1000 | Otos 1 | Otos 2 | Otos 3 | Otos 4 | Otos 5 | Keskiarvo |
| Aika (ms) | 10,00 | 10,02 | 10,00 | 10,30 | 10,29 | 10,12 |
| Jätekuorma (kb) | 0,18 | 0,18 | 0,18 | 0,18 | 0,18 | 0,18 |